Hardware Efficient PDE Solvers in Quantized Image Processing

Vom Fachbereich Mathematik
der Universität Duisburg-Essen
(Campus Duisburg)

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
genehmigte Dissertation von

Robert Strzodka
aus Tarnowitz

Referent: Prof. Dr. Martin Rumpf
Korreferent: Prof. Dr. Thomas Ertl
Datum der Einreichung: 30 Sep 2004
Tag der mündlichen Prüfung: 20 Dez 2004

Contents

Abstract

Performance and accuracy of scientific computations are competing aspects. A close interplay between the design of computational schemes and their implementation can improve both aspects by making better use of the available resources. The thesis describes the design of robust schemes under strong quantization and their hardware efficient implementation on data-stream-based architectures for PDE based image processing.

The strong quantization improves execution time, but renders traditional error estimates useless. The precision of the number formats is too small to control the quantitative error in iterative schemes. Instead, quantized schemes which preserve the qualitative behavior of the continuous models are constructed. In particular for the solution of the quantized anisotropic diffusion model one can derive a quantized scale-space with almost identical properties to the continuous one. Thus the image evolution is accurately reconstructed despite the inability to control the error in the long run, which is difficult even for high precision computations.

All memory intensive algorithms are, nowadays, burdened with the memory gap problem which degrades performance enormously. The instruction-stream-based computing paradigm reenforces this problem, whereas architectures subscribing to data-stream-based computing offer more possibilities to bridge the gap between memory and logic performance. Also more parallelism is available in these devices. Three architectures of this type are covered: graphics hardware, reconfigurable logic and reconfigurable computing devices. They allow to exploit the parallelism inherent in image processing applications and apply a memory efficient usage. Their pros and cons and future development are discussed.

The combination of robust quantized schemes and hardware efficient implementations deliver an accurate reproduction of the continuous evolution and significant performance gains over standard software solutions. The applied devices are available on affordable AGP/PCI boards, offering true alternatives even to small multi-processor systems.

AMS Subject Classification (MSC 2000)

- **65Y10** Numerical analysis: Algorithms for specific classes of architectures
- **68U10** Computer science: Image processing

ACM Computing Classification System (CSS 1998)

- **G.4** Mathematical Software: Efficiency, Reliability and robustness, Parallel and vector implementations
- **I.4.3** [Image Processing and Computer Vision]: Enhancement—Smoothing, Registration
- **I.4.6** [Image Processing and Computer Vision]: Segmentation—Region growing and partitioning
- **G.1.8** [Numerical Analysis]: Partial Differential Equations—Finite element methods, Finite difference methods, Parabolic equations, Hyperbolic equations, Multigrid and multilevel methods
- **B.3.1** [Memory Structures] Semiconductor Memories—Dynamic memory (DRAM)
- **I.3.1** [Computer Graphics]: Hardware Architecture—Graphics processors
- **B.7.1** [Integrated Circuits]: Types and Design Styles—Gate arrays
- **C.1.3** [Processor Architectures]: Other Architecture Styles—Adaptable architectures
- **C.4** Performance of Systems: Performance attributes
- **J.3** Life and Medical Sciences: Health

General Terms: Algorithms, Languages, Performance, Theory

Keywords

quantization, qualitative error control, quantized scale-space, memory gap, performance, data-stream-based processing, graphics hardware, reconfigurable logic, reconfigurable computing

1 Introduction

Contents

Tables

The motivation section presents the broader picture of the thesis and outlines ideas which embrace the different chapters. In the Thesis Guide we present a quick overview of the thesis and cover presentational aspects. The chapter ends with a summary of the results and acknowledgments.

1.1 Motivation

Numerical mathematics is concerned with the design of fast and accurate schemes for the approximate solution of mathematical problems. Computer systems are the target platforms for the implementation of theses schemes. So the trade-off between the competing factors of performance and accuracy applies both to the mathematical level where approximations of different accuracy order are chosen, and the implementational level where number formats and operations of different precision are used. Traditionally, the optimization processes are performed separately by mathematicians and computer scientists respectively. The common interface is the operation count of a scheme which is sought to be reduced. We argue that this measure is much too simple as it completely ignores the diverse performance characteristics of computer systems. Thus apparently efficient mathematical schemes perform surprisingly badly on actual systems. In the area of partial differential equation (PDE) based image processing the thesis demonstrates how an early consideration of performance relevant hardware aspects and a close coupling of the scheme design and its implementation fully exploit the available resources and so deliver fast and accurate solutions.

1.1.1 Operation Count and Performance

The merits of the exponential development in semiconductors have benefited memory and computing elements in different respects. Data transport and data processing have not developed at the same pace. The consequences are far-reaching but can be outlined by an example.

If we consider a simple addition of two vectors $\bar{C} = \bar{A} + \bar{B}$ of size N, then the operation count is N. Modern micro-processors can process two operands made up of four 32-bit floats in one clock cycle. So if the processor runs at 3GHz it can perform 12G floating point OPS (FLOPS) and we should finish the addition in $N/12$ ns. Real performance values are at least an order of magnitude lower. The processor can really run almost 12G FLOPS if not disturbed by anything else, but the data cannot be transported that quickly. The parallel operations namely require 96GB/s of input data and 48GB/s for the output. But the current memory systems can provide a *bandwidth* of at most 6.4GB/s. This means that the computational unit spends 95% of time waiting for the data. So global data movement and not local computations are expensive and decisive for the overall performance.

The situation becomes even worse when the components of the vectors $\bar{A}, \bar{B}$ are not arranged one after another in memory, e.g. if they are part of larger structures or arranged in irregular lists. Then memory *latency*, the time needed to find the individual components in memory, becomes dominant and the performance can drop by as much as an order of magnitude again. Therefore, it is often advisable to enforce a linear arrangement of vector components, even if this means the inclusion of additional entries to fill up the gaps of the irregular arrangement. The operation count is increased, but the data can be processed in a seamless data stream, avoiding the latencies.

These two components data addressing and data transport dominate the execution times of many algorithms. This fact has been acknowledged for some time already, and remedies have been developed, but the problem grows with each generation of new processors. Meanwhile hardware architectures subscribing to a data oriented computing paradigm have evolved. We evaluate three different representatives of this concept on image processing applications. The results show that the focus on regular data handling instead of minimal operation count delivers superior results.

1.1.2 Precision and Accuracy

In image processing applications performance is very critical. For this purpose one is often prepared to sacrifice strict accuracy requirements as long the quality of the results does not suffer significantly. The question arises how can we secure robust results with less precise computations.

A number format has only finitely many representations for the real numbers in a computer. So beside the error introduced by the discretization of the continuous PDE models, we also have an additional error associated with the quantized number representation and the approximate computations. Thus the quality of the final result depends both on the precision of the quantization and the properties of the schemes. It is very dangerous to trust in high precision alone. The simple computation $1 - 1.0002 \cdot 0.9998$ in the single float format, for example, evaluates to zero, although the correct result $4 \cdot 10^{-8}$ can be represented exactly. Even long double float formats do not save us from these problems (see Section 2.2.2.1 on page 27).

But for the sake of performance we want to operate on much smaller number formats, e.g. $8-16$ bit. The mathematical challenge is to design schemes which can still deliver satisfactory results in this setting. For one iteration of a scheme strict error bounds can be derived. But these bounds avail to nothing if we iterate the scheme up to several hundred times. Since the number of iterations is sometimes even larger than the number of representable values in the number format (256 for 8 bit), all hope concerning accuracy seems to be lost. In fact, accuracy in the usual sense of error bounds which quantitatively relate the computed to the continuous solution cannot be obtained for such low precision. But the quality of the results depends not necessarily on this quantitative relation, but rather the qualitative behavior of the PDE model. So the guiding idea is to preserve invariants and characteristics of the evolution of the continuous models. These properties depend more on the appropriate design of the quantized schemes than the precision of the number formats. So despite a lack of control on the quantitative error we obtain accurate results in the sense of reproduced qualitative properties.

1.1.3 Choice of Applications and Architectures

We have chosen the field of image processing, because it favors the use of parallel computations, which are widely used by our architectures, and allows the qualitative computational approach discussed in the previous section. We intentionally did not restrict ourselves to the study of discrete algorithms, but chose the PDE based models to demonstrate that the continuous properties can be preserved even with low precision computations. Having the continuous models in the background has also the advantage that from their analysis one knows a-priori which qualitative properties of the evolution are desirable and how they are influenced by the parameters. There is no dependence on the discrete nature of a computing model at the continuous stage, so that all such effects can be more easily identified in the implementation.

The choice of architectures has been motivated by the focus on processing in data streams as discussed in Section 1.1.1 on page 2. We evaluate three different architectures: graphics hardware, reconfigurable logic and reconfigurable computing devices. Their way to performance gains is similar: massive parallelism and efficient memory usage, but the programming models are quite different. To fortify the practicability of the presented approach to image processing, we have deliberately selected architectures which are available on affordable AGP or PCI accelerator boards and can be used with a standard Personal Computer (PC). The processing capabilities of graphics hardware even come for free, since every current PC already contains a powerful Graphics Processor Unit (GPU).

Throughout the thesis we argue strongly for the early consideration of hardware characteristics in the design of numerical solvers. However, this does not mean that we favor machine-intimate, low level languages over standardized High Level Languages (HLLs). Although Hardware Description Languages (HDLs) offer more opportunities to utilize the full potential of the hardware, they have several disadvantages concerning the design effort, code reuse, compatibility and maintenance. We do not advocate the change from the currently predominant extreme of 'hardware blind' programming, to the other extreme of low level HDLs. The primary advantages of data oriented architectures can be exploited with high level considerations about the arrangement of data, memory access and the use of parallel processing elements (PEs). In most of our implementations we had to resort to low level languages only because standard HLLs are basically blind to hardware aspects and hardware efficient HLLs are yet in the development. We hope that these new HLLs will soon allow to solve problems of much higher complexity with full hardware support.

Despite the high parallelism of the architectures and good applicability to the image processing tasks we have restricted the implementations to 2D data sets. The PDE models and the quantized schemes can be extended to 3D easily. The implementations require reorganization to a different extent for 3D, but basically no new implementational ideas are involved. The reason why we do not present 3D examples, is our conviction that adaptive methods are indispensable for three dimensions. Even the massive parallelism of our architectures cannot compete with adaptive methods if data volume grows cubically. Naturally, adaptivity in 2D can also gain performance, but because of the trade-off between operation count and regular data access (Section 1.1.1 on page 2) the advantages are less striking. Initial work on hardware efficient adaptive 3D methods has already been performed and will be continued, but it is an extensive topic on its own facing many new challenges and therefore is not part of this thesis.

1.2 Thesis Guide

This is a quick guide dealing with issues which concern the thesis as a whole.

1.2.1 Thesis Structure

The chapters begin with a list of contents, figures and tables, and a sketch of the discussed topics. The last section in each chapter contains a detailed summary of the themes with accurate references to the previous sections. It may serve as a good overview for someone familiar with the subject. For a first orientation we give very concise information on the contents and prerequisites of the following chapters. For a summary of the results we refer to Section 1.3 on page 8 at the end of this chapter.

- 2. PDE Solvers in Quantized Image Processing
  Here we introduce the PDE models for image processing and perform analysis of the properties of the discrete, quantized solvers. The chapter assumes a general understanding of PDEs and Finite Element (FE) methods. It is fairly independent of the other material. Mathematicians not interested in the reasoning about the implementation may want to proceed, after the quantized scheme analysis in each section, directly to the corresponding result sections in Chapter 4.

- 3. Data Processing
  This chapter explains the reasons for the *memory gap* problem and how the different computing paradigms deal with it. It describes the suitability of various hardware architectures for image processing algorithms and thus motivates the selected devices used in Chapter 4. The chapter is basically self-contained and assumes only very basic knowledge about computer systems.

- 4. Hardware Efficient Implementations
  Chapter 4 picks up the quantized schemes from Chapter 2 and discusses their efficient implementation under the considerations from Chapter 3. For readers interested mainly in the implementations it is probably best to get first an idea of the continuous model properties in Section 2.1 on page 15, and then continue with the corresponding implementation section in Chapter 4. The chapter does not assume familiarity with the non-standard architectures used, but for those new to this subject we recommend reading Chapter 3 for a broader picture and a better understanding.

1.2.2 Index and Acronyms

The index contains a list of key words. Referenced occurrences of these terms appear in *italic* in the text. ***Bold italic*** marks the main reference position for a term. At this position the meaning and context of the term can be found.

Table 1.1 General notation.

Symbol	Example	Explanation
u, ϕ	$u(x) = x^2$	continuous functions
U, Φ	$U(x) = \sum_\alpha \bar{U}_\alpha \Theta_\alpha(x), \quad (\Theta_\alpha)_\alpha$ basis	discrete functions
$\bar{U}, \bar{\Phi}$	$\bar{U}_\alpha = (\bar{U}_{(0,0)}, \ldots, \bar{U}_{(N_x-1,0)}, \ldots, \bar{U}_{(N_x-1,N_y-1)})^\top$	nodal vectors
α, β	$\alpha = (\alpha_x, \alpha_y) = (1, 2)$	2D multi-indices
$\mathcal{Q}$	$\mathcal{Q} = \{\frac{n}{255} \vert n = 0, \ldots, 255\}$	quantized number system
$\oplus, \ominus, \odot, \oslash$	$\bar{V} \oplus \bar{W}$	quantized arithmetic
$\mathrm{eval}^{\mathcal{Q}}(\mathrm{term})$	$\mathrm{eval}^{\mathcal{Q}}(\bar{V} - \bar{W}) = \bar{V} \ominus \bar{W}$	quantized evaluation
$=_{\mathcal{Q}}$	$\sqrt{\frac{1}{2}} =_{\mathcal{Q}} \frac{180}{255}$	quantized evaluation
$g(.)$	$g(x) = \exp(-c_g x)$	functions
$f[.]$	$f[u](x) = \int_0^x u(y)dy$	operators
Symbol	Definition	Explanation
$\mathbb{1}$	$\mathbb{1}(x) := x, \ (\mathbb{1}\bar{U})_\alpha := \bar{U}_\alpha$	continuous, discrete identity
$\bar{0}, \bar{1}$	$\bar{0} := (0, \ldots, 0)^\top, \ \bar{1} := (1, \ldots, 1)^\top$	zero and one vector
$\#$	$\#I := \ \min_{f: I \to \mathbb{N} \text{ injective}} \max f(I)$	number of elements in a set
δ	$\delta_{\alpha\beta} := \{1 \text{ if } \alpha = \beta; \ 0 \text{ else}\}$	Kronecker symbol
diag	$\mathrm{diag}(L) := \ (\delta_{\alpha\beta} L_{\alpha\beta})_{\alpha\beta}$	diagonal of a matrix
supp	$\mathrm{supp}\ (u) := \{x \in \Omega \vert u(x) \neq 0\}, u : \Omega \to \mathbb{R}$	support of a function

Acronyms are written out in the long form at least once, when they appear first in the text. Their meaning is also explained in an alphabetical listing on page 201 just before the index.

1.2.3 Notation

Table 1.1 summerizes the general notation. Hereby, most continuous quantities involved in the PDE models are typed in small letters, e.g. u. The corresponding discrete functions are set in capital letters (U). The nodal vectors defining the discrete functions are marked by a bar ($\bar{U}$) and addressed ($\bar{U}_\alpha$) by two dimensional multi-indices, e.g. $\alpha = (\alpha_x, \alpha_y) \in \mathbb{N} \times \mathbb{N}$. For further details we refer to Section 2.2.1.2 on page 26.

The nodal vectors are generally assumed to be already quantized elements of some quantization $\mathcal{Q}$. Quantized arithmetic operations are denoted by $\circledcirc \in \{\oplus, \ominus, \odot, \oslash\}$ (Section 2.2.3.1 on page 36). Quantized evaluations of terms are written as $\mathrm{eval}^{\mathcal{Q}}(\mathrm{term})$ or $=_{\mathcal{Q}}$ (Section 2.2.3.2 on page 38).

Table 1.2 International decimal and binary prefixes.

Decimal factor	Decimal name	Decimal symbol	Binary factor	Binary name	Binary symbol
10^3	kilo	k	2^{10}	kibi	Ki
10^6	mega	M	2^{20}	mebi	Mi
10^9	giga	G	2^{30}	gibi	Gi
10^{12}	tera	T	2^{40}	tebi	Ti
10^{15}	peta	P	2^{50}	pebi	Pi
10^{18}	exa	E	2^{60}	exbi	Ei

We distinguish between functions $g(u(x))$ and operators $f[u](x)$ with different brackets. The first case is actually a composition of functions $(g \circ u)(x)$ and represents the value of the function g at the position $u(x)$. In the second case f is an operator which takes the function u as an argument and returns a new function $f[u]$ as a result. This new function is then evaluated at x.

For clarity, we keep up this distinction in the discrete case, e.g. $G(\bar{V}), L[\bar{V}]$, although a discrete operator could also be seen as a high dimensional function of the finitely many vector components.

1.2.4 Binary Prefixes

There is a confusion about the use of the Système International d'Unités (SI)-prefixes in computer science, where typically the binary system is used and byte (B) or bit (b) quantities are multiples of 2. When the referred numbers were small the capitalization of $1\text{kB} = 1000\text{B}$ to $1\text{KB} = 1024\text{B}$ provided an appropriate distinction, but this does not apply to the larger prefixes, e.g. 1MB has been used for both $1,000,000$B and 1024^2B $= 1,048,576$B depending on the context.

Since 1999 there exists an international standard on the naming of 2^{10n} factors [IEC, 2000]. Table 1.2 presents the decimal and binary symbols and names. Initially the standard has been generally ignored, but the approval of the symbols by the Linux developers in 2001 and the trial-use announcements by the Institute of Electrical and Electronics Engineers (IEEE) and American National Standards Institute (ANSI) in 2002 have helped to widen its popularity. We make use of the handy distinction throughout the work and also translate quantities from other sources into this terminology. Where the sources are not totally clear which prefixes were meant we have made a choice to the best of our knowledge.

Due to the organization of memory (see Section 3.1.1 on page 71), sizes of memory chips, caches and sizes of data objects stored therein use almost exclusively the binary factors, while *bandwidth* and *throughput* are expressed with the decimal factors, because they are based on frequencies given in MHz or GHz. For mass storage devices the situation is inconsistent with the capacity of hard discs, DVDs and most other disk or tape devices being given in the decimal system, while the dimensions of CDs and flash memory based devices (USB sticks, CompactFlash cards, etc.), and the file sizes in operating systems are calculated in the binary system, but displayed misleadingly with the decimal factors. The confusion is complete for floppy disks where '1.44MB' means neither 1.44MB nor 1.44MiB but $1,440$KiB, which is twice the capacity of old 720KiB disks and derived as $512\text{B}(\text{sector size}) \cdot 9(\text{sectors/track}) \cdot 80(\text{tracks}) \cdot 2(\text{sides}) = 720\text{KiB}$. Although current disk and tape devices use also binary sector sizes like 1KiB or 4KiB for the smallest storable data block, similar to the floppy disk their arrangement does not depend on powers of two, so that the more marketing friendly decimal factors are used.

1.3 Summary

The main contribution of the thesis is the close coupling of the numerical scheme design with the implementation. Careful treatment of roundoff behavior in the discrete PDE solvers secures qualitatively accurate results despite strong quantization, and thorough consideration of the performance characteristics of the hardware architectures delivers high performance. The numerical and hardware aspects, which are usually dealt with separately by mathematicians and computer scientists, receive here an integral treatment to the benefit of both performance and accuracy.

The following sections summarize the individual chapters. For a very concise overview and prerequisites of the chapters see Section 1.2.1 on page 5.

1.3.1 PDE Solvers in Quantized Image Processing

We deal with three important application areas of image processing: denoising, segmentation and registration (Section 2.1 on page 15). The solution to these problems is often needed in real-time, such that the requirements on performance are very high. Each task is modeled by a PDE and an approximate, numerical solution can be obtained after time and space discretization (Section 2.2 on page 24). Bounds on the difference between the continuous and discrete solution exist, but these bounds assume error-free computations. In real computer systems quantization as a third source of error comes into play.

It is very important to realize that even high precision floating point formats such as long double (s63e15) can easily lead to completely wrong results for simple computations (Section 2.2.2.1 on page 27). In PDE solvers many iterative computations must be performed, so that one cannot trust in high precision number formats alone. Deterministic, probabilistic and empirical methods can be used to derive bounds for the quantization errors of algorithms. But because of the very high performance requirements, image processing algorithms tend to use low precision fixed point number systems (Section 2.2.2.2 on page 30). For these systems it is impossible to guarantee numerical stability of iterative schemes in neither deterministic nor probabilistic sense (Section 2.2.2.3 on page 33). In practice, empirical evidence from error simulators and test suits is used on a case by case basis to estimate the effects of quantization for a given algorithm. But this is a time consuming and intransparent procedure. Moreover, the empirical evidence cannot guarantee the desired behavior for all possible input data.

The main result of Chapter 2 is the design of discrete schemes which can guarantee a certain behavior of the quantized PDE solvers despite strong quantization. The key idea is to preserve the desired global properties of the continuous model rather than try to control the accuracy of individual computations. Hereby, the factorization and aggregation of numbers on different scales and the interactions between different node values deserve special attention (Sections 2.3.1 on page 43, 2.4.1 on page 55, 2.5.1 on page 59). In particular, a careful implementation of the matrix vector product as a main ingredient of the discrete schemes is decisive. Involved symmetric matrices have often unit column sums ($\forall j : \sum_i A_{ij} = 1$) preserving the overall mass (sum of the vector components) in a matrix vector product (Eq. 2.26 on page 39). In low precision arithmetic underflows can easily violate the mass preservation and other global properties, and the iterative nature of the algorithm quickly accumulates these errors to produce visual artefacts (Figure 4.8 on page 140). The *mass-exact matrix vector product* (Section 2.2.3.3 on page 39) guarantees the mass preservation irrespective of the used quantization. Moreover, it is well suited for the field of image processing as it operates with highest accuracy around edges (Section 2.2.3.5 on page 42).

When used in the *anisotropic diffusion* scheme the mass-exact matrix vector product also secures other important properties of the quantized scale-space (Section 2.3.3 on page 49), most notably the extremum principle (Eq. 2.53 on page 51) and Lyapunov functionals (Eq. 2.57 on page 52). The derivation of the quantized scale-space is a very satisfactory result, as it inherits almost all of the properties of the continuous scale-space of the anisotropic diffusion operator (Section 2.1.1.2 on page 17). In particular, the decrease of energy, central moments and the increase of entropy follow for arbitrary quantizations.

In case of the *level-set equation* used for segmentation we have put the focus on the best possible resolution of different velocities during the evolution of the interface. Clearly the quantization restricts the number of distinct velocities, but scaling schemes can secure that the faster moving parts evolve with the highest available precision. Moreover, the stationary asymptotic behavior equals the continuous model (Section 2.4.3 on page 58).

The multi-scale regularization of the registration problem makes the quantized scheme very robust against quantization errors. Cutting the precision of the used number format in half hardly changes the results (Section 4.1.5.2 on page 147). This is also achieved by the robust diffusion schemes used in the regularization at various stages. So despite the high complexity of the algorithm and numerous intermediate result stages, the quality of the low precision results is not corrupted by roundoff errors.

The design of the robust schemes partly depends on a space discretization with an equidistant grid (Section 2.2.1.2 on page 26). This impedes dynamic adaptivity, but trading operation count for a regular memory access pattern (Section 1.1.1 on page 2) is often advantageous, because the regular data-flow of the image processing applications (Section 2.6 on page 63) makes them suitable for highly parallel architectures.

1.3.2 Data Processing

The exponential growth of transistors on the same area of silicon has influenced the characteristics of memory and computing logic differently (Section 3.1 on page 71). This is not a necessity but economical forces steer the development to a great extent. With smaller feature sizes the timings of PEs and the size of Dynamic RAM (DRAM) chips have risen exponentially, but the timings of the memory core which stores the 0s and 1s have improved in comparison very little (Section 3.1.1.1 on page 72). An order of magnitude lies between each of the times needed for the addition of two numbers, their transport to the processor and their localization in the memory in case of a random memory access.

The problem of diverging memory and logic performance, labeled *memory gap*, grows each year, and from an economic point of view there is no remedy in sight. Current improvements benefit mainly *bandwidth*, while in comparison *latency* even worsens (Section 3.1.1.2 on page 76). For optimal performance almost all latencies can be hidden, but this requires a predictable data-flow of the algorithm. Processing of data in streams and maximal data reuse in memory hierarchies (Section 3.1.3 on page 81) have become very beneficial for performance.

In view of the memory gap, Chapter 3 irradiates different computing paradigms (Section 3.2 on page 84), parallelization options (Section 3.2.2 on page 87), the status quo of current hardware architectures (Section 3.3.1 on page 97) and the future evolution of computing machines (Section 3.3.2 on page 103). The resource utilization of multiple PEs with different types of *parallelism* and the minimization of bandwidth and latency requirements play a decisive role in these sections. But in many cases the availability of high level programming tools rather than the characteristics of the hardware matter for the economic success of an architecture. The easy serial programming model has mainly prevailed over performance considerations so far, but the sheer number of new architectures (Sections 3.2.4 on page 93, 3.2.5 on page 96), many of which are available as commercial products, indicates that the performance gains are too large to be still ignored. The quantized schemes for the image processing tasks from Chapter 2 are particularly suitable for the acceleration on these new devices.

1.3.3 Hardware Efficient Implementations

Three data-stream-based (DSB) architectures have been used for the implementation of the image processing applications: GPUs, a Field Programmable Gate Array (FPGA) and the eXtreme Processing Platform (XPP). The distribution of the solvers on the architectures reflects their availability and ease of operation. GPUs are the most common and inexpensive, and have been used for several implementations. Availability of FPGAs is high, but not so much the costs of the hardware but those of the development tools pose higher barriers to common usage. Finally, the coarse-grain XPP array has only been available for a limited period of time as a clock cycle exact simulator. But now a PCI card with that device can be purchased.

The early implementations of the diffusion models and the level-set equation in graphics hardware (Sections 4.1.3 on page 127 and 4.1.4 on page 132) demonstrated the applicability of GPUs as scientific coprocessors and inspired other work in that area. The *virtual signed 16 bit format* (Figure 4.10 on page 141) was the first to overcome the 8 bit limitation of GPUs in general, providing more suitability for scientific computations (Figure 4.11 on page 142). On newer hardware more advanced numerical concepts such as multi-scale representation, multi-grid solvers and adaptive time-step control have been realized for the registration problem (Section 4.1.5 on page 143). Although the other architectures offer even higher resource utilization, the GPU has now the advantage of a high level access to its functionality which greatly facilitates the programming. The complex algorithm for the registration would be very difficult to code on a FPGA or the XPP.

The FPGA implementation of the level-set equation exploits the full flexibility of the low level reconfigurability (Section 4.2.3 on page 161). Application specific data paths and operations, and *variable precision arithmetic* bring the small device (approx. one third of a DX8 GPU) in lead of much larger and higher clocked processors. More recent FPGAs can further increase performance of such algorithms by utilizing embedded hardwired multipliers, which otherwise consume a lot of configurable resources.

The size of the XPP64-A1 architecture is comparable to a DX8 GPU (approx. 50M transistors), but its flexible access to massive parallelism beats the competitors (Section 4.3.3 on page 175). Similar to the FPGA the whole scheme is merged into a fully pipelined super computation unit which delivers a result pixel in each clock cycle.

Despite the low precision number formats available on the devices, the special quantized schemes from Chapter 2 allow to retain the main properties of the continuous models. At the same time the implementations can outperform a comparable micro-processor by factors of 5 to 20. Thus the goal of fast low precision computations with preserved global properties has been reached. The current hardware development and the better scalability of the architectures suggest that these factors will grow further in future. Concurrently, the complexity of the problems will increase and the availability of hardware efficient High Level Language will gain even more relevance for the broad success of these architectures.

Acknowledgments

First of all my thanks goes to my advisor Martin Rumpf who encouraged me to work on this interdisciplinary topic and supported me in the sometimes difficult advancement of the research. I am particularly grateful for his initiative towards the use of different hardware architectures, which lead to a broader and more abstract perspective on the topic.

During the work I have received plenty of help from my colleagues and the administration of the Institute of Applied Mathematics at the University of Bonn and the Numerical Analysis and Scientific Computing group at the University of Duisburg-Essen. In Bonn, Ralf Neubauer and Michael Spielberg were very helpful in designing the first implementations, Thomas Gerstner worked with me on volume rendering and Jochen Garcke often helped with computer problems. In the difficult beginning the former PhD students Martin Metscher and Olga Wilderotter offered advice. Both in Bonn and Duisburg the development of the applications was expertly supported by Tobias Preusser, Marc Droske and Ulrich Clarenz. Udo Diewald was an unebbing source of helpful tips and information, and Ulrich Weikard, my room mate in Duisburg, was always ready for an inspiring discussion. He and Martin Lenz proof-read and discussed the thesis with me. Martin's scientific vicinity to parts of the thesis also helped me with some complex improvements of it. Special mention deserves Birgit Dunkel, our ever helpful secretary in Duisburg. Many thanks to all my colleagues for their different contributions to the success of this work.

I am also grateful to many persons of other facilities who have helped me in various ways. The successful research with Oliver Junge and Michael Dellnitz from the University of Paderborn brought about my first scientific publication and fortified my decision towards this thesis. The German National Academic Foundation supported me during my PhD allowing me to participate in many stimulating discussions. Heinz-Josef Fabry from the Faculty of Catholic Theology at the University of Bonn was my consultant and gave me much insight outside of the mathematical universe. The work of Matthias Hopf and Thomas Ertl from the University of Stuttgart made me aware of the looming computing potential in graphics hardware. The fruitful discussions with Matthias also helped me with the peculiarities of early graphics hardware. Markus Ernst, Steffen Klupsch and Sorin Huss from the Technical University of Darmstadt introduced me to the world of Reconfigurable Logic. Markus and Steffen directly contributed to the thesis by coding the hardware configuration for the level-set equation on a FPGA. The company PACT XPP Technologies from Munich granted me a temporary free research license to work with their XPP architecture and Prashant Rao readily answered my questions concerning its programming. In late 2002 I joined the caesar research center in Bonn directed by Karl-Heinz Hoffmann, who has generously supported my further research. Although the recent results are not directly included in this thesis my coworkers have positively influenced its contents. I am indebted to Marcus Magnor and Ivo Ihrke from the Max Planck Institute for Computer Science in Saarbrücken, Alexandru Telea from the Technical University of Eindhoven, Aaron Lefohn and John Owens from the University of California, Davis, Ross Whitaker from the University of Utah, Salt Lake City, and Christoph Garbe from the University of Heidelberg for inspiring discussions.

In such long lasting work not only the scientific but also the social support is of great importance. Special thanks to my friends and above all my parents and brothers and sisters who accompanied me through the highs and lows. Their constant encouragement made this work possible.

2 PDE Solvers in Quantized Image Processing

Contents

Figures

This chapter contains the theoretical analysis of different PDE based image processing algorithms with respect to quantization. For a given quantization every operation can be performed up to a certain precision. The aggregation of this effect is similar to the effects of time and space discretization of the continuous models and accordingly we are interested in convergence and approximation behavior. In particular for fixed point numbers, which exhibit large relative error in many computations, we find equivalent reformulations of the discrete solvers which reproduce their qualitative behavior even under strong quantization.

The problems of denoising, segmentation and registration are modeled by different differential equations, but their discretizations focus on the creation of a common regular data-flow. The choice of discretization is also influenced by the quantization since appropriate re-ordering of operations in the schemes allows to retain a higher accuracy. By paying attention to such issues the developed solvers unveil their sensitivity to hardware architectures and the performance benefits are discussed in Chapter 4 on page 107.

2.1 Continuous PDE Based Image Processing

Images undergo many modifications on the way from the acquisition to the output device. A typical series of processing stages includes:

- Reconstruction of the image data from the raw data of the acquisition device.
- Removal of unavoidable noise in the image due to the acquisition process.
- Identification of segments in the image which are of special interest to the application.
- Correlation of the image information with other images of similar content.
- Visualization of the image with regard to the characteristics of the human visual system.

Each of these tasks has drawn a lot of research attention. The first is a little less accessible, since the manufacturers of image acquisition devices usually know the peculiarities of their hardware best and already include the reconstruction into the device. The last task on the other hand is a very extensive field of its own, since the required form of data presentation varies widely from application to application.

Therefore, we restrict ourselves to the three central tasks of image denoising, segmentation and registration. Still there have been numerous ways of approaching these problems. We concentrate on continuous PDE models which encode the nature of the problem implicitly in functional equalities. This general choice of methods is motivated in Section 1.1.3 on page 3 in the introduction. The following sections review the PDE models associated with the three afore mentioned image processing tasks.

2.1.1 Denoising - Diffusion Processes

Image data often contains noise stemming from the acquisition or transmission process. If certain characteristics of the noise distribution are known, one usually incorporates appropriate filters into the acquisition or transmission process itself. In the absence of other a-priori knowledge, noise is often assumed to be normally distributed. A very simple but effective method for reducing such noise is the application of a Gaussian filter, which is related to a linear diffusion process. Unfortunately this technique also reduces highly relevant edge information in the image. Non-linear methods which aim at distinguishing between relevant image information which should be protected and noise fluctuations which should be removed are necessary for a feature sensitive denoising. The anisotropic diffusion model represents a class of these non-linear methods.

2.1.1.1 Generic Diffusion Model

We consider the function $u : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ in the domain $\Omega := [0,1]^2$ and an initial noisy image given as a function $u_0 : \Omega \to \mathbb{R}$. The idea is to evolve the initial image through a partial differential equation such that in the sequence of the resulting images $u(t,.), t > 0$ the noise dissolves while edges are preserved. The ***generic diffusion*** model reads:

$$\begin{aligned} \partial_t u - \operatorname{div}\left(G(\nabla u_\sigma)\nabla u\right) &= 0\ , && \text{in } \mathbb{R}^+ \times \Omega\,, \\ u(0) &= u_0\,, && \text{on } \Omega\,, \\ G(\nabla u_\sigma)\nabla u \cdot \nu &= 0\ , && \text{on } \mathbb{R}^+ \times \partial\Omega\,, \end{aligned} \tag{2.1}$$

where $G : \mathbb{R} \to \mathbb{R}^{2\times 2}$ is the ***diffusion tensor*** depending on a mollification u_σ of u.

We may distinguish three major choices for G:

- ***Linear diffusion*** model $G(\nabla u_\sigma) := \mathbb{1}$
  In this case we have the linear heat equation:

  $$\partial_t u - \Delta u \;=\; 0\,.$$

  The solution can be obtained by convolution (denoted by '*')

  $$u(t) = \mathcal{G}_{0,\sqrt{2t}} * u_0,$$

  with a ***Gaussian function***

  $$\mathcal{G}_{\mu,\sigma}(y) \;:=\; (2\pi\sigma^2)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}\frac{(y-\mu)^2}{\sigma^2}\right)\,. \tag{2.2}$$

  The parameter μ is not required here, but will be needed later in a different context (Eq. 2.6 on page 20). According to this model the intensities in the image will dissolve homogenously in every direction and with equal rate at every position. This model is used for fast and simple smoothing and regularization, e.g. the mollification u_σ may be performed with the linear model:

  $$u_\sigma \;:=\; u(\sigma^2/2) \;=\; \mathcal{G}_{0,\sigma} * u_0\,.$$

- ***Non-linear diffusion*** model $G(\nabla u_\sigma) := \tilde{g}(\|\nabla u_\sigma\|)\,\mathbb{1}$
  Here the diffusion tensor depends on a non-negative, monotone decreasing scalar function $\tilde{g} : \mathbb{R} \to \mathbb{R}^+_0$. A typical choice for $\tilde{g}$ is the ***Perona-Malik function***

  $$\mathcal{P}_{c_{\tilde{g}}}(x) \;:=\; \frac{1}{1 + c_{\tilde{g}} x^2} \tag{2.3}$$

  or an exponential equivalent like $\exp(-c_{\tilde{g}}x)$, such that for large gradient modulus $\|\nabla u_\sigma\|$ the function $\tilde{g}$ has small values meaning small diffusion rate, and for small gradient modulus large values meaning high diffusion rate. The mollification u_σ helps to distinguish edges from singular high gradients due to noise. Consequently the diffusion rate in this model depends non-linearly on the modulus of the gradient of intensities at each position, but the diffusion direction is homogenous.

- ***Anisotropic diffusion*** model $G(\nabla u_\sigma) := B(\nabla u_\sigma)^\top g(\|\nabla u_\sigma\|) B(\nabla u_\sigma)$
  This extends the previous model to a direction dependent diffusion rate. Now g is a diagonal matrix with independent diffusion coefficients for the direction along the smoothed gradient and orthogonal to it, while B is the transformation from the vectors in the canonical basis to the vectors in the gradient-normal basis:

$$B(\nabla u_\sigma) := \frac{1}{\|\nabla u_\sigma\|} \begin{pmatrix} \partial_x u_\sigma & \partial_y u_\sigma \\ -\partial_y u_\sigma & \partial_x u_\sigma \end{pmatrix}, \qquad (2.4)$$
$$g(\|\nabla u_\sigma\|) := \begin{pmatrix} g_1(\|\nabla u_\sigma\|) & 0 \\ 0 & g_2(\|\nabla u_\sigma\|) \end{pmatrix}.$$

  Typically g_1 is chosen as the Perona-Malik function $\mathcal{P}$ like in the previous case and $g_2(x) = c_{g_2}$ is some positive constant. In contrast to the previous, this model can smooth distorted edges because diffusion will only be inhibited in the direction orthogonal to the edge, while the smoothing along the edge will still take place in the non-linear fashion.

2.1.1.2 Scale-Space

The solution $u : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ can be regarded as a multi-scale of successively diffused images $u(t), t \geq 0$, which is called the ***scale-space*** of u_0. We want to study the properties of this scale-space in the form of the corresponding scale-space operator S_t, which is defined as

$$S_t[u_0] \quad := \quad u(t).$$

See Section 1.2.3 on page 6 concerning the square bracket notation. S_t is thus the solution operator of the diffusion problem. We list several interesting properties of this operator. Details and proofs can be found in [Weickert, 1998], but many of the properties are straight-forward consequences of the fact that the diffusion process is in divergence form and the diffusion tensor $G(\nabla u_\sigma)$ depends solely on ∇u_σ. The following properties hold true for an arbitrary function $v \in L^2(\Omega)$, all $x \in \mathbb{R}$ and all $t, s \geq 0$ unless otherwise stated.

- Semigroup property

$$\begin{aligned} S_0 &= \mathbb{1}, \\ S_{t+s} &= S_t \circ S_s\,. \end{aligned}$$

- Grey level shift invariance

$$\begin{aligned} S_t[0] &= 0, \\ S_t[v+c] &= S_t[v] + c, \ \text{for } c \in \mathbb{R}. \end{aligned}$$

- Reverse contrast invariance

$$S_t[-v] \quad = \quad -S_t[v]\,.$$

- Average grey level invariance
$$\mathcal{M}\big[S_t[v]\big] \;=\; \mathcal{M}[v]\,,$$
where $\mathcal{M} : L^1(\Omega) \to L^1(\Omega)$ is the averaging operator defined by
$$\left(\mathcal{M}[v]\right)(x) \;:=\; \frac{1}{|\Omega|}\int_\Omega v(y)\;dy\,.$$

- Translation invariance
$$(S_t \circ \tau_p)[v] \;=\; (\tau_p \circ S_t)[v]\,,$$
for any translation $(\tau_p[v])(x) := v(x+p)$ with supp $\tau_p[v]$, supp $(\tau_p \circ S_t)[v] \subseteq \Omega$.

- Isometry invariance
$$(S_t \circ R)[v] \;=\; (R \circ S_t)[v]\,,$$
for any orthogonal transformation $R \in \mathbb{R}^{2\times 2}$ defined by $(R[v])(x) := v(Rx)$ with supp $R[v]$, supp $(R \circ S_t)[v] \subseteq \Omega$.

- Extremum principle
$$\operatorname*{ess\,inf}_\Omega v \;\le S_t[v(x)] \le\; \operatorname*{ess\,sup}_\Omega v\,.$$

- Lyapunov functionals
For $v \in L^2(\Omega)$ and $r \in C^2[\operatorname{ess\,inf}_\Omega v, \operatorname{ess\,sup}_\Omega v]$ with $r'' \ge 0$, the functional
$$\Phi[t,v] \;:=\; \int_\Omega r\left(S_t[v(x)]\right)\;dx$$
is a ***Lyapunov functional***:
$$\begin{aligned}\Phi[t,v] &\ge \Phi[0,\mathcal{M}[v]] \quad \forall t>0,\\ \Phi[.,v] &\in C[0,\infty)\cap C^1(0,\infty),\\ \partial_t\Phi[t,v] &\le 0 \quad \forall t>0.\end{aligned}$$

If $r'' > 0$ on $[\operatorname{ess\,inf}_\Omega v, \operatorname{ess\,sup}_\Omega v]$ then $\Phi[t,v]$ is a strict Lyapunov functional:
$$\begin{aligned}\Phi[0,v] = \Phi[0,\mathcal{M}[v]] \quad&\Longleftrightarrow\quad v = \mathcal{M}[v] \text{ a.e. on } \Omega,\\ \partial_t\Phi[t,v] = 0 \iff \Phi[t,v] = \Phi[0,\mathcal{M}[v]] \quad&\Longleftrightarrow\quad S_t[v] = \mathcal{M}[v] \text{ on } \Omega \quad \forall t>0,\\ \Phi[0,v] = \Phi[t,v] \text{ for } t>0 \quad&\Longleftrightarrow\quad \begin{cases} v = \mathcal{M}[v] \text{ a.e. on } \Omega \text{ and}\\ S_s[v] = \mathcal{M}[v] \text{ on } \Omega, \forall s \in (0,t].\end{cases}\end{aligned}$$

- Convergence
$$\lim_{t\to\infty}\|S_t[v]-\mathcal{M}[v]\|_p \;=\; 0 \text{ for } p \in [1,\infty).$$

We have listed these properties in such detail, because later we want to show that we can preserve many of this continuous properties in equivalent form in the discrete-quantized setting (Section 2.3.3 on page 49).

2.1.1.3 Related Work

The non-linear diffusion models as we know them today were first introduced by a work of Perona and Malik [Perona and Malik, 1990]. Their model denoises images while retaining and enhancing edges. But analysis of the Perona-Malik model showed its mathematical ill-posedness [Kawohl and Kutev, 1998; Kichenassamy, 1997; You et al., 1996]. A regularized model [Catté et al., 1992] was derived, which can still preserve edges for an appropriately chosen regularization parameter. A thorough rigorous axiomatic theory of these methods under the term of scale-spaces was given in [Alvarez et al., 1993]. Different choices of the non-linearity and especially the use of the so called structure tensor for direction sensitive smoothing of images were presented in [Weickert, 1998].

2.1.2 Segmentation - Level-Set Methods

The distribution of intensities in an image is usually perceived as a collection of objects by humans. Our visual system performs the task of segmenting the data into regions of certain characteristics. The fact that these regions often represent known objects simplifies the task. But even when meaning cannot be assigned, we still distinguish regions by certain features and sometimes even complete them in mind despite the absence of the corresponding information in the image. The human low level task of finding homogenous regions of some characteristic is coupled with the recognition of objects. Computer segmentation, however, seeks to perform the low-level task using appearance criteria only. The main difficulty lies in the appropriate encoding of the target objects' characteristics into these criteria. A very flexible method to account for these characteristics is the controlled expansion of a curve. Starting with a small circle such a curve is supposed to expand up to the borders of the segment in which it lies. Different object characteristics can be encoded by steering the expansion velocity of the curve with the local image values, their derivatives and the form of the curve. The level-set method is an implicit formulation of this expansion process, which naturally incorporates curve splitting and merging.

2.1.2.1 Curve Evolution

We are given the original image $p : \Omega \to \mathbb{R}$ on $\Omega := [0,1]^2$ and the initial curve $C_0 : [0,1] \to \Omega$. We represent this curve implicitly as the zero level-set $\{x \in \Omega \big| \phi_0(x) = 0\}$ of a continuous function $\phi_0 : \Omega \to \mathbb{R}$ defined on the whole domain Ω. Then we ask for a solution $\phi : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ to the *level-set equation*

$$\begin{aligned} \partial_t \phi + f^\sigma[\phi] \cdot \nabla \phi &= 0 \;, \quad \text{in } \mathbb{R}^+ \times \Omega \,, \\ \phi(0) &= \phi_0 \,, \quad \text{on } \Omega \,, \end{aligned} \tag{2.5}$$

where $f^\sigma[\phi]$ is the velocity field which drives the evolution of the level-set function ϕ. In case of a complex velocity field with varying directions on the boundary it is not obvious how to define boundary conditions for ϕ. Following [Sethian, 1999] we impose a mirror boundary condition for ϕ on $\mathbb{R}^+ \times \partial\Omega$. The velocity field is usually composed of three different forces

$$f^\sigma[\phi] \quad = \quad f_g^\sigma \frac{\nabla\phi}{\|\nabla\phi\|} + f_\kappa[\phi]\frac{\nabla\phi}{\|\nabla\phi\|} + f_V\,,$$

where f_g^σ and $f_\kappa[\phi]$ are forces in the normal direction of the level-sets and f_V is independent thereof. The square brackets denote operators, see Section 1.2.3 on page 6. The forces have different impact on the evolution:

- External forces $f_g^\sigma(t,x) = c(t,x) + g_1(p(x)) + g_2(\|\nabla p_\sigma(x)\|)$.
  The first term $c(t,x)$ can prescribe a default expansion speed and will usually be simply a constant value. By using a characteristic function χ_R it can also favor the expansion in an a-priori known region of interest R. The other terms take into account local information of the image p and may take the following form:

  (2.6)
  $$\begin{aligned} g_1(y) &:= \mathcal{G}_{\mu,c_{g_1}}(y) = (2\pi {c_{g_1}}^2)^{-\frac{1}{2}} \exp\left(-\frac{1}{2}\frac{(y-\mu)^2}{{c_{g_1}}^2}\right), \\ g_2(y) &:= \mathcal{P}_{c_{g_2}}(y) = \frac{1}{1+c_{g_2}y^2}, \end{aligned}$$

  with the similar use of the *Gaussian function* $\mathcal{G}_{\mu,c_{g_1}}$ and the *Perona-Malik function* $\mathcal{P}_{c_{g_2}}$ as for the diffusion processes in Section 2.1.1.1 on page 15. The effect is to forward expansion in areas with intensities similar to μ and small mollified gradients. The mollification controlled by σ prevents too high sensitivity of the gradient to noise. Such evolution produces segments with smoothly distributed intensity around μ. Depending on the choice of c_{g_1} and c_{g_2} the similarity to μ or the smoothness of the segment is more relevant.

- Curvature $f_\kappa[\phi](t,x) = -c_\kappa \kappa[\phi](t,x)$.
  The curvature κ in two dimensions is given as

  $$\kappa[\phi] \quad := \quad \nabla \cdot \frac{\nabla\phi}{\|\nabla\phi\|} \quad = \quad \frac{\phi_{xx}\phi_y^2 - 2\phi_x\phi_y\phi_{xy} + \phi_{yy}\phi_x^2}{(\phi_x^2+\phi_y^2)^{3/2}},$$

  where subscripts indicate partial derivatives here. The inclusion of this term will hamper the formation of high curvature bends in the level-sets. This is especially useful in preventing the propagation front from expanding through a narrow hole in an almost complete segment border. By this one wants to mimic to a certain degree the human ability for form completion. The general effect is an evolution of smoother level-sets.

- Velocity field $f_V(t,x) = V(t,x)$.
  Level-set methods are often used in the tracking of phase boundaries between different materials or material states. In these applications the whole system is usually subjected to forces which act on the interface independent of its form. Therefore, the possibility of an external velocity field interfering with the evolution is given. But also in the case

of segmentation such a field may be useful to penalize the deviation of the segment from an a-priori given default form. This feature could also help to complete a segment in areas where the boundary information is missing in the data.

The solution $\phi : \mathbb{R}^+ \times \Omega \to \mathbb{R}$ of the problem represents a family of successively evolved level-sets $L(t, c) := \{x \in \mathbb{R}^2 | \phi(t, x) = c\}$, $c \in \mathbb{R}$. The curves $L(t, 0), t \geq 0$ describe the evolution of our initial curve $C_0 = L(0, 0)$. In the asymptotic state $t \to \infty$ the curve $L(t, 0)$ converges to an equilibrium of forces or expands over the borders of the image. For applications, however, we are interested only in a finite and preferably short evolution time T, such that the process is stopped as soon as the changes in the level-sets become very small.

2.1.2.2 Hamilton-Jacobi Equation

By introducing the Hamilton function $H(u) := f_p \|u\| + f_V \cdot u$ we may reformulate the level-set equation in the form of a Hamilton-Jacobi equation

$$\partial_t \phi + H(\nabla \phi) \quad = \quad c_\kappa \kappa[\phi] \|\nabla \phi\|.$$

with a parabolic viscosity term on the right hand side. Further setting $u := \nabla \phi$ and differentiating the above equation leads to the hyperbolic conservation law:

$$\partial_t u + \nabla(H(u)) \quad = \quad c_\kappa \nabla \left(\|u\| \nabla \frac{u}{\|u\|} \right). \tag{2.7}$$

Because of their importance in physics conservation laws and their numerical implementations have been studied thoroughly [Corrias et al., 1995]. Therefore, this alternative perspective on the level-set equation helps us later to find the appropriate discretization.

2.1.2.3 Related Work

Active contour models which describe the evolution of a initial curve driven by the underlying image, external data and curve form have become a very popular tool in image segmentation [Osher and Sethian, 1988]. The development started with explicitly parameterized curve representations. But the key disadvantage of this method is a topological constraint: the curve cannot split to approximate boundaries of not simply connected segments. Such problems have been solved by introducing implicit models [Caselles et al., 1993; Malladi et al., 1995], in which the initial curve is interpreted as the zero level-set of a function defined on the whole domain. The evolution of this function is controlled by a PDE [Sethian, 1999]. Sensitivity of the process to initialization has been later reduced by an appropriate choice of driving forces [Xu and Prince, 1998]. Furthermore a general variational framework for Mumford-Shah and Geman type functionals [Geman et al., 1990; Mumford and Shah, 1985] has been introduced [Hewer et al., 1998], where edge boundaries are represented by a discontinuous function, yielded by the minimization of an energy functional.

2.1.3 Registration - Energy Gradient Flows

Images often come as a series of exposures of the same object from different times or perspectives, sometimes even acquired by different imaging technologies. The series are used for the analysis of temporal changes or the overall structure of the object. The analysis requires an information correlation between two images. A correlation is ideally given by a deformation of the images which matches the object representation of the first image onto that of the second. The quality of the deformation depends on the definition of optimal matching criteria.

For a clearly arranged configuration the human visual system quickly suggest an appropriate deformation. But without any semantic knowledge about the content of the images in the computer, the matching criteria must rely solely on the image intensities and their geometric distribution and therefore often do not suffice to pinpoint the optimal deformation, especially when the images contain a diffuse distribution of intensities; then we are faced with a multitude of possible solutions and must introduce some reasonable assumptions on the deformation to distinguish among them. Energy gradient flows allow a very flexible way to define and control these additional assumptions which are crucial in the search for a reasonable image correlation.

2.1.3.1 Energy Model

Given two images, a template and a reference $T, R : \Omega \to \mathbb{R}$, $\Omega \subset \mathbb{R}^2$, we look for a deformation $\phi : \Omega \to \Omega$ which maps the intensities of T via ϕ to the intensities of R such that

$$T \circ \phi \quad \approx \quad R\,.$$

Since ϕ is usually small in comparison to $|\Omega|$ it can be suitably expressed as $\phi = \mathbb{1} + u$, with a displacement function u. The displacement u is sought as the minimum of the energy

$$E[u] \quad = \quad \frac{1}{2}\int_\Omega |T \circ (\mathbb{1} + u) - R|^2\,.$$

The square brackets are used for operators, see Section 1.2.3 on page 6.

A minimizer u in some Banach space $\mathcal{V}$ is characterized by the condition

$$\int_\Omega E'[u] \cdot \theta = 0\,,$$

for all $\theta \in [C_0^\infty(\Omega)]^2$, with the L^2-representation of E'

$$E'[u] \quad = \quad (T \circ (\mathbb{1} + u) - R)\nabla T \circ (\mathbb{1} + u)\,,$$

if we assume T to be smoothly differentiable. This gradient may be used as the descent direction towards a minimum in a gradient descent method. But there may be many minima since any displacements within a level-set of T do not change the energy. To guarantee that the gradient descent converges to a unique solution, we have therefore to exclude solutions which contain irregular mappings within the level-sets of T by applying a regularization technique. The descent along the gradient will be regularized by $A(\sigma)^{-1}$, with

$$A(\sigma) \;=\; \mathbb{1} - \frac{\sigma^2}{2}\Delta,\; \sigma \in \mathbb{R}^+ .$$

Then the regularized gradient flow

$$\begin{aligned} \partial_t u &= -A(\sigma)^{-1}E'[u]\,, && \text{in } \mathbb{R}^+ \times \Omega\,, \\ u(0) &= u_0 \qquad\qquad\;, && \text{on } \Omega\,, \end{aligned} \tag{2.8}$$

has a unique solution u with $u(t) \in \mathcal{V}$ for some function space $\mathcal{V}$ (Theorem 3.1 [Clarenz et al., 2002]). In the implementation the regularization with $A(\sigma)^{-1}$ can be quickly realized by a multi-grid cycle with few smoothing steps, because we do not require the exact solution but rather the smoothing properties.

2.1.3.2 Multi-Scale Hierarchy

Since the energy E is non-convex the gradient descent path may easily get trapped in local minima instead of finding the global minimum of E. Therefore, a continuous annealing method is used by defining a multi-scale of image pairs

$$\begin{aligned} T_\epsilon &:= S(\epsilon)T, \\ R_\epsilon &:= S(\epsilon)R, \end{aligned} \tag{2.9}$$

for $\epsilon \geq 0$ with a filter operator S. The choice $S(\epsilon) = A(\epsilon)^{-1}$ corresponds again to a Gaussian filtering. The energy

$$E_\epsilon[u] = \frac{1}{2}\int_\Omega |T_\epsilon \circ (\mathbb{1} + u) - R_\epsilon|^2$$

induces the corresponding gradient flow on scale ϵ, which has the solution u_ϵ.

For the annealing process we choose an exponentially decreasing series of scales $(\epsilon_k)_{k=0,\ldots,K}$, $K \in \mathbb{N}$ with $\epsilon_0 = 0$ and use the approximate solution of the gradient flow problem from a coarser scale ϵ_k at a sufficiently large time t_{ϵ_k} as the initialization for the gradient flow problem on a finer scale ϵ_{k-1}, i.e. we perform:

$$\begin{aligned} u_{\epsilon_K}(0) &:= 0, \\ u_{\epsilon_{k-1}}(0) &:= u_{\epsilon_k}(t_{\epsilon_k})\,, \end{aligned}$$

for $i = K-1, \ldots, 1$ until the final solution $u_{\epsilon_0}(t_{\epsilon_0})$ on the finest scale ϵ_0 is reached. In the implementation the multi-scales can be efficiently encoded in a multi-grid hierarchy.

2.1.3.3 Related Work

If we measure image coherence by the energy $E[u] = \frac{1}{2}\int_\Omega |T \circ \phi - R|^2$, where T, R are the intensity maps of the images and ϕ the deformation, then the minimization problem is ill-posed, because arbitrary deformations within the level-sets of T do not change the energy. Therefore, many regularizations of the registration problem have been discussed in the literature [Christensen et al., 1997; Davatzikos et al., 1996; Grenander and Miller, 1998; Maes et al., 1997; Thirion, 1998]. The registration approach given above was introduced in [Clarenz et al., 2002]. But the ideas are similar to the iterative Tikhonov regularization methods [Hanke and Groetsch, 1998], fast multi-grid smoothers [Henn and Witsch, 2001] and multi-scale use for large displacements [Alvarez et al., 2000] presented previously.

2.2 Discretization - Quantization

In this section we develop a common setting for the discretization and quantization of the continuous models. The similar treatment offers advantages for the analysis of the discrete algorithms and already pays attention to hardware issues by explicitely exposing the inherent parallelism of the algorithms. The similar structure of the algorithms is also important for the reuse of the same architecture for different image processing tasks.

Concerning the quantization we give a short overview of different number formats, their shortcomings with respect to accuracy in computations and attempts to remedy these problems. Then we concentrate on methods which help to conserve the qualitative behavior of the continuous models even under quantization that is so coarse that it defeats the direct methods of error enclosure or statistical control. Error analysis of this methods shows also the limitations of this approach.

2.2.1 Discrete Operators

Although the introduced models for denoising, segmentation and registration are fairly different they have the common form:

$$\begin{aligned} \partial_t u + \mathcal{F}[\mathcal{C}^p_\sigma[u], u] &= 0 \;, \quad \text{in } \mathbb{R}^+ \times \Omega\,, \\ u(0) &= u_0\,, \quad \text{on } \Omega\,, \end{aligned} \tag{2.10}$$

with an additional condition on $\mathbb{R}^+ \times \partial\Omega$ for the *generic diffusion* model and the *level-set equation* which involve spatial derivatives of the unknown. Here, $\mathcal{C}^p_\sigma[u]$ and $\mathcal{F}[u, v]$ are operators on the Banach space $\mathcal{V}$. For the discussed problems they read:

(2.11)

problem	equation	$\mathcal{C}_\sigma^p[u]$	$\mathcal{F}[u,v]$
denoising	Eq. 2.1 on page 16	$G(\nabla u_\sigma)$	$-\text{div}\,(u\nabla v)$
segmentation	Eq. 2.5 on page 19	$f^\sigma[u]$	$u \cdot v$
registration	Eq. 2.8 on page 23	$A(\sigma)^{-1}E'[u]$	u

The non-linear operator $\mathcal{C}_\sigma^p[u]$ serves as a local ***classifier***, i.e. the function u is locally classified with respect to the desired evolution. The indices p and σ indicate that the classifier may depend on additional input data p and regularization parameters σ, which control the data sensitivity of the process. The result of the operator can be considered as a weight function, which determines the magnitude of impact of the regions of u at the current time t on the evolution of u. The operator $\mathcal{F}[u,v]$ performs the application of these weights. It may still involve spatial derivatives, but has to be linear in u and either also linear in v or independent of v.

The unified PDE scheme for the problems (Eq. 2.10 on the facing page) helps us to define a common discretization approach.

2.2.1.1 Time Discretization

We use explicit and implicit schemes for time discretization. The simple explicit Euler gives:

$$u^{n+1} = u^n - \tau^n \cdot \mathcal{F}[\mathcal{C}_\sigma^p[u^n], u^n]. \tag{2.12}$$

This schemes has the advantage of fast execution and allows good control of quantization effects but is quite restricted in the time-step width τ^n. The ***Courant-Friedrichs-Levy (CFL)*** condition requires that

$$\|\tau^n \cdot \mathcal{F}[\mathcal{C}_\sigma^p[u^n], u^n]\|_\infty < h\,, \tag{2.13}$$

if h is the element grid size in the spatial discretization, i.e. information is prohibited to cross a whole grid cell in one time-step. We will use the explicit discretization for the level-set method in the segmentation problem and the gradient descent in the registration problem.

The fully implicit Euler scheme

$$u^{n+1} + \tau^n \cdot \mathcal{F}[\mathcal{C}_\sigma^p[u^{n+1}], u^{n+1}] = u^n\,, \tag{2.14}$$

would give us unconditional stability, but the implicit treatment of the non-linearities in $\mathcal{C}_\sigma^p$ would require very sophisticated and computationally intensive solvers. Instead, we take a middle course by treating the linear terms implicitly and the non-linearities explicitely. This semi-implicit scheme gives us good stability and leads to a linear instead of a non-linear equation system after spatial discretization:

$$u^{n+1} + \tau^n \cdot \mathcal{F}[\mathcal{C}_\sigma^p[u^n], u^{n+1}] = u^n\,.$$

We will use this scheme for some of the diffusion models.

2.2.1.2 Space Discretization

An equidistant quad mesh underlies our spatial discretizations. The reason for this choice lies in the tremendous parallelization and pipelining opportunities resulting from this simple structure. Certainly, the missing adaptivity requires a finer overall grid size h, but the simply structured algorithms can be mapped onto the hardware architectures so well, that the gained performance almost always outweighs computing time associated with the additional degrees of freedom. Moreover, input data in image processing naturally comes as rectangular images without any geometry beneath. In case where input data comes from simulations on complicated, possibly unstructured grids, one may consider a fast resampling in the hardware onto a Cartesian grid before the processing [Westermann, 2001].

A hardware efficient approach to adaptivity is the division of the domain into blocks, where each block contains an equidistant grid, or more generally a deformed tensor grid. The domain is then processed block by block such that each time the operations are performed on a regular array of data and only boundary values must be appropriately exchanged. This technique provides a transition layer from fully adaptive meshes to hardware accelerated adaptive computing and with the appropriate choice of parameters can be used for different hardware architectures [H.Becker et al., 1999]. For parallel computers it has the additional advantage that the regular form of the blocks facilitates domain decomposition and load balancing. In the following we do not develop such a layer but rather concentrate on the optimal discretization and quantization of the individual blocks.

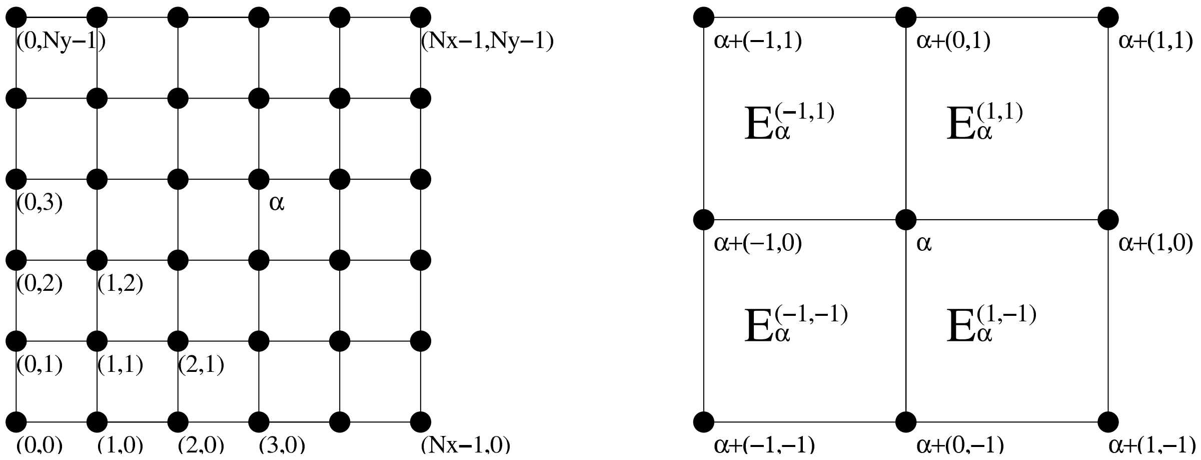
The domain $\Omega = [0,1]^2$ is discretized by a $N_x \times N_y$ equidistant grid Ω_h with grid size h. The nodes of the grid are indexed by a 2-dimensional multi-index $\alpha = (\alpha_x, \alpha_y) \in (0, \dots, N_x - 1) \times (0, \dots, N_y - 1)$. This grid is used as a basis of either Finite Difference or Finite Element schemes. The bilinear discrete functions, e.g. $U : \Omega \to \mathbb{R}^+_0$, are denoted by capital letters. The images of size (N_x, N_y) are represented by the corresponding nodal vectors, e.g. $(\bar{U}_\alpha)_\alpha$ with $\bar{U}_\alpha := U(h \cdot \alpha)$. Figure 2.1 on the next page shows the indices of the grid, and the local situation with local index offsets $\gamma := \beta - \alpha$ and the neighboring elements E^γ_α for a given node α.

2.2.2 Quantization

Quantization refers to the representation of a continuous quantity by a finite set of values. The values are often equally distributed over the range of the continuous quantity. Certainly, we commit an error in such a process. In digital systems quantization error arises from the quantization of the input data, the quantization of the problem parameters and roundoff errors, i.e. quantization of intermediate results. Because quantization is a necessity, both the data [Gray and Neuhoff, 1998] and the arithmetic error [Bornemann et al., 2004; Higham, 2002] have been studied intensely, and the references above cover a wide range of topics and point to an extensive additional literature.

Although there are software packages which offer number formats of arbitrary precision or

Figure 2.1 On the left an equidistant $N_x \times N_y$ grid enumerated by a tuple, on the right the neighboring elements of a node and the local offsets to neighboring nodes.



algebraic number representations one almost always opts for a compromise between accuracy and performance, because time is often a critical factor in applications. In particular in image processing performance often has highest priority and the quality of results depends on global features rather than exact computations in detail. Because quantization of the analog signals occurs during the acquisition process of images we will be mainly concerned with the round-off error in the processing. Depending on the actual quantization the effects of the roundoff as the cause of error in arithmetic operations and the interaction of these effects can be quite different. In the following we will consider the most common quantizations with fixed precision, namely the floating point and the fixed point number formats, their shortcomings and their contributions to numerical stability.

2.2.2.1 Floating Point Numbers

Floating point formats are the most common number representations in scientific computing. Their great advantage is that the separate representation of an exponent allows to cover a large number range with relatively few bits. A ***floating point number*** system $\mathcal{Q}^{\text{FP}}(\beta, t, e_{\text{min}}, e_{\text{max}}) \subset \mathbb{Q}$ is given by

$$\mathcal{Q}^{\text{FP}}(\beta, t, e_{\text{min}}, e_{\text{max}}) := \left\{ \pm m \cdot \beta^{e-t} \middle| m \in [0, \beta^t - 1] \cap \mathbb{N}, e \in [e_{\text{min}}, e_{\text{max}}] \cap \mathbb{Z} \right\}, \tag{2.15}$$

where the variables have the following meaning:

- β, the base or radix.
  It defines the base of the number system used for the floating point numbers, typically $\beta = 2$.
- e, the ***exponent***.
  It describes the scale on which a number resides. The range of the exponent $[e_{\text{min}}, e_{\text{max}}]$

depends on the number of bits reserved for its representation and the number of special cases, e.g. $e_{\text{min}} = -126, e_{\text{max}} = 127$ in the IEEE-754 single precision format. The exponent is represented as an unsigned 8 bit $\bar{e} \in [0, 255]$ biased by -127. The value $\bar{e} = 0$ is reserved for denormalized numbers (see below), and $\bar{e} = 255$ for infinite or undefined numbers.

- m, the significand or ***mantissa***.
  It represents the equidistantly spaced floating point numbers on a given scale,
- t, the ***precision***.
  It describes the fineness of the number spacing on a given scale.

To ensure a unique representation one requires that for non-zero numbers $\beta^t > m \geq \beta^{t-1}$ holds, and speaks of a ***normalized representation***. In case of base 2, the most significant bit in m in the normalized representation is always one and thus needs not to be stored, one says that the ***hidden bit*** is 1. Many floating point number systems support so called ***denormalized numbers*** with a hidden bit 0 indicated by a special value of the exponent. Denormalized numbers represent additional floating point numbers which lie between zero and the smallest normalized number. We will not further elaborate on these special cases indicated by reserved exponent values.

A typical example of a fixed precision floating point format is the IEEE-754 s23e8 single precision format with 1 sign bit, 23 bits of mantissa and 8 bits for the exponent. The format uses the base $\beta = 2$ and has the precision $t = 24$, which is one larger than the number of bits reserved for the mantissa, because of the hidden bit.

We define the increment operator and the decrement operator on $\mathcal{Q}$ which define the succeeding or preceeding number respectively:

$$\begin{aligned} \text{inc}^{\mathcal{Q}}(x) &:= \min\{q \in \mathcal{Q} \big| q > x\}, \\ \text{dec}^{\mathcal{Q}}(x) &:= \max\{q \in \mathcal{Q} \big| q < x\}\,. \end{aligned}$$

Then $\text{inc}^{\mathcal{Q}}(0)$ is the smallest representable positive non-zero number and $\text{inc}^{\mathcal{Q}}\left(\text{inc}^{\mathcal{Q}}(0)\right)$ the next one. Now the question arises how the distances between succeeding numbers vary and the following definition gives the answer. The ***machine epsilon*** $\varepsilon^{\mathcal{Q}}$ is defined as the distance from 1.0 to the next larger floating point number

$$\varepsilon^{\mathcal{Q}} := \text{inc}^{\mathcal{Q}}(1.0) - 1.0\,, \tag{2.16}$$

and since 1.0 is represented as $\beta^{t-1} \cdot \beta^{1-t}$ we have

$$\varepsilon^{\mathcal{Q}} = \beta^{1-t}\,.$$

This quantity describes the spacing of floating point numbers on a given scale, i.e. numbers on the scale e lie in $[\beta^{e-1}, \beta^{e})$ and have the equidistant spacing $\varepsilon^{\mathcal{Q}} \cdot \beta^{e-1} = \beta^{e-t}$, e.g. numbers on the scale 1 lie in $[1.0, \beta)$ and have the spacing $\varepsilon^{\mathcal{Q}}$.

The general question how good we can approximate real numbers with the floating point number system $\mathcal{Q}$ across all scales is answered by the following quantity. The ***unit roundoff*** is the minimal upper bound on the relative error in approximating a real number:

$$\begin{aligned} u^{\mathcal{Q}} &:= \min\left\{ u \in \mathbb{R} \middle| \forall x \in W^{\mathcal{Q}} : \exists q \in \mathcal{Q} : \frac{|x-q|}{|x|} < u \right\}, \\ W^{\mathcal{Q}} &:= \mathbb{R} \cap [\min \mathcal{Q}, \max \mathcal{Q}] \setminus [\mathrm{dec}^{\mathcal{Q}}(0), \mathrm{inc}^{\mathcal{Q}}(0)]. \end{aligned} \tag{2.17}$$

Since $\frac{|x-q|}{|x|}$ is invariant to scaling, we can rescale an arbitrary $x \in W^{\mathcal{Q}} \setminus \mathcal{Q}$ by powers of β such that $\tilde{x} \in (1.0, \beta)$, if $x \in \mathcal{Q}$ then the relative error is zero. The floating point numbers in $[1.0, \beta)$ are equidistantly spaced with $\varepsilon^{\mathcal{Q}}$ and therefore there is a $\tilde{q} \in \mathcal{Q}$ such that $|\tilde{x} - \tilde{q}| \le \frac{1}{2}\varepsilon^{\mathcal{Q}}$. The inequality $\frac{|\tilde{x}-\tilde{q}|}{|\tilde{x}|} < \frac{1}{2}\varepsilon^{\mathcal{Q}}$ becomes strict, because $\tilde{x} \in (1.0, \beta)$. Thus we have

$$u^{\mathcal{Q}} = \frac{1}{2}\varepsilon^{\mathcal{Q}} = \frac{1}{2}\beta^{1-t}. \tag{2.18}$$

It is important to realize that floating point numbers only control the relative error $\frac{|x-q|}{|x|}$ very well, but the absolute error $|x-q|$ varies strongly from as little as $\frac{1}{2}\beta^{e_{\min}-t}$ to as much as $\frac{1}{2}\beta^{e_{\max}-t}$. Most of the error analysis of floating point arithmetic therefore relies on the assumption:

$$x \circledcirc y = (x \circ y)(1+\varepsilon), \quad \circ \in \{+, -, \cdot, /\},\ |\varepsilon| \le u^{\mathcal{Q}},$$

where $\circledcirc$ denotes the corresponding approximate operation in floating point numbers. This means that the approximate result of the computation $x \circledcirc y$ has a relative error ε bounded in the same way as if the result was exactly computed and then rounded to the nearest floating point number.

There exists a considerable number of results deduced from this assumption and we refer to [Higham, 2002] for an extensive discussion of them. Although computations in common floating point units fulfill stronger assumption on the properties of floating point operations, these additional properties are seldomly used to prove better error bounds [Priest, 1992]. But despite the fact that the machine results are often more accurate than the rigorous estimates predict, the finite precision arithmetic still bears some vicious roundoff effects. In the following we sketch some of the problems in floating point roundoff.

We consider the function

$$f(x,y) := (333.75 - x^2)y^6 + x^2(11x^2y^2 - 121y^4 - 2) + 5.5y^8 + x/(2y) \tag{2.19}$$

at $x_0 = 77617, y_0 = 33096$. This is a modified version [Loh and Walster, 2002] of Rump's function [Rump, 1988]. A straight-forward C++ program compiled with the GNU Compiler 2.95.3 on a Pentium 4 Linux system with the powers expanded to multiplications gives:

$$\begin{aligned} &\text{float s23e8} && f(x_0,y_0) =_{\mathcal{Q}} 1.1726, \\ &\text{double s52e11} && f(x_0,y_0) =_{\mathcal{Q}} 1.17260394005318, \end{aligned}$$

$$\text{long double s63e15} \qquad f(x_0, y_0) =_{\mathcal{Q}} 1.172603940053178631\,,$$

where "$=_{\mathcal{Q}}$" denotes the evaluation under the respective quantization $\mathcal{Q}$. We are inclined to trust in the result, since increased precision only appends further decimal positions. But all results are wrong and even the sign is wrong. A variable precision interval arithmetic tool [Ely, 1990] encloses the result in the interval

$$\begin{aligned}[\quad & -0.827396059946821368141165095479816292005, \\ & -0.827396059946821368141165095479816291986 \quad].\end{aligned}$$

Because floating point numbers work very well most of the time, we do not expect such sudden difficulties, even though the unavoidable rounding with roundoff errors as the actual cause of such problems is clear to us. The erroneous results stem from the handicap of fixed precision floating point format in which the exponent allows to represent numbers on many scales but their interference leads to roundoff errors, which can be dramatically enlarged by cancellation, e.g. for s24e8 floats we have:

$$\begin{aligned} \text{additive roundoff} \qquad & a = 10^8 + 4 =_{\mathcal{Q}} 10^8, \\ \text{multiplicative roundoff} \qquad & b = (10^4 + 2) \cdot (10^4 - 2) =_{\mathcal{Q}} 10^8, \\ \text{cancellation } c \in \{a, b\} \qquad & (c - 10^8) \cdot 10^8 =_{\mathcal{Q}} 0\,. \end{aligned}$$

We see how cancellation easily promotes the absolutely (4) and relatively $(4 \cdot 10^{-8})$ small additive and multiplicative errors to an absolute error of $4 \cdot 10^8$ and an infinite relative error. We also notice that the order of operations may play an important role in computations. Although in the above example changing the order to $(10^8 - 10^8 + 4) \cdot 10^8$ easily solves the problem, there is no general rule for avoiding the cancellation in concurrent additions [Higham, 1993]. However, despite the unavoidable roundoff one can still obtain reliable results with computer arithmetic. So called ***enclosure methods*** can provide rigorous bounds for the result of an algorithm, including all intermediate rounding errors. They often use *directed rounding modes*, *exact dot products* and *variable precision arithmetic* to achieve this goal. An overview of the literature on these verified numerical results can be found in [Bohlender, 1996]. In our case of low precision implementations we would need unreasonably many computations to prevent the interval bounds derived in such methods from expanding to the whole number range.

We will use floating point arithmetic in the registration problem, but not so much because of the exponent but rather because of a lack of higher precision fixed point formats in graphics hardware. Fixed point formats may be seen as a special case of floating point formats with zero bits for the exponent. Then, all numbers are represented on the same scale and the problems with roundoff errors become rather different.

2.2.2.2 Fixed Point Numbers

Fixed point computations require only integer arithmetic and are therefore extremely fast and efficient in terms of hardware resource requirements. However, they represent numbers on a single scale and therefore quickly suffer from over- or underflow. Similar to the floating point number system we define the ***fixed point number*** system $\mathcal{Q}^{\text{FX}}(\beta, t, \varepsilon) \subset \mathbb{Q}$ as

$$\mathcal{Q}^{\text{FX}}(\beta, t, \varepsilon) \quad := \quad \{m \cdot \varepsilon \big| m \in [-\beta^t, \beta^t - 1] \cap \mathbb{Z}\}, \tag{2.20}$$

with β as the base, t the precision and ε the scale. The numbers discretize the interval $[-\beta^t \cdot \varepsilon, (\beta^t - 1) \cdot \varepsilon]$ in equidistant steps of ε. In analogy to the floating point format (Eq. 2.16 on page 28) we will call this quantity associated with the quantization $\mathcal{Q}$ the *machine epsilon* $\varepsilon^{\mathcal{Q}}$ and we have two major choices:

- $\varepsilon^{\mathcal{Q}} = \beta^{e-t}$.
  The exponent $e < t$ gives the number of positions in the integral part, and $t - e$ gives the number of positions in the fractional part. The represented numbers have the form $m \cdot \beta^{e-t}$ and contain the floating point numbers with precision t on scale e, but each lower scale $e_1 \in [e - t + 1, e - 1]$ is represented with the lower precision $t - (e - e_1)$.

  The problem with this format is the missing representation of the integer β^e. When we want to cover the interval $[-1, 1]$ (i.e. $e = 0$), this means that 1.0 cannot be represented. In floating point formats, this is not a problem, because 1.0 is represented on the next scale, but here we have only one scale and choosing $e = 1$ would mean that we reduce the precision on scale 0 for the interval $[-1, 1]$ by a factor of β.

- $\varepsilon^{\mathcal{Q}} = 1/(\beta^{t-e} - 1)$.
  This format intends to overcome the problem of non-representation of β^e, especially in the case $\beta^0 = 1.0$, at the cost of a non-power of two machine epsilon. It has the disadvantage that very few fractions can be represented exactly ($\beta^{t-e} - 1$ has usually few factors) and rounding in multiplications is more complicated. This choice leads also to a precision dependent over-representation outside of the integral interval $[-\beta^e, \beta^e]$, producing overflows at artificial values. But one can clamp the result to the above interval, and moreover in the most common case $e = 0$ there is only one over-represented number namely $-\beta^t/(\beta^t - 1)$.

  The great advantage of this format is definitely the exact representation of $-\beta^e, 0, \beta^e$, in particular $-1, 0, +1$ in the special case $e = 0$. The representation of the neutral elements of addition, multiplication and its inverse guarantees that in repeated additions and multiplications certain areas of the computational domain can remain unchanged or only change the sign, without having to specify a different operation on them.

In contrast to the small relative error of approximating real numbers by the floating point number system (Eq. 2.18 on page 29), the *unit roundoff* $u^{\mathcal{Q}}$ for a fixed point quantization is quite irrelevant, because the choice $x = \frac{3}{2}\varepsilon^{\mathcal{Q}}$ in Eq. 2.17 on page 29 implies that

$$u^{\mathcal{Q}} \quad > \quad \frac{1}{3}\,. \tag{2.21}$$

This means that fixed point formats have a very bad relative error $\frac{|x-q|}{|x|}$ in the approximation of real numbers and instead we should consider the absolute error $|x - q|$.

Obviously we can define a mapping $\text{round}^{\mathcal{Q}}_{\mu} : \mathbb{R} \to \mathcal{Q}$ called rounding function, such that for all $x \in \mathbb{R} \cap [\min \mathcal{Q}, \max \mathcal{Q}]$ we have

$$|\text{round}^{\mathcal{Q}}_{\mu}(x) - x| \;<\; \varepsilon^{\mathcal{Q}} .$$

The subscript $\mu \in \mathfrak{d} \cup \mathfrak{n}$ indicates the different ***rounding modes***. The ***directed rounding modes*** $\mathfrak{d} := \{0, \infty, -\infty, +\infty\}$ with rounding towards zero, rounding towards infinity, rounding towards minus infinity, rounding towards plus infinity, and the ***nearest rounding modes*** $\mathfrak{n} := \{n\!\uparrow, n\!\downarrow, n2\}$ are defined by

(2.22)
$$\begin{aligned} -\varepsilon^{\mathcal{Q}} &< |\text{round}^{\mathcal{Q}}_{0}(x)| - |x| &\leq 0, \\ \varepsilon^{\mathcal{Q}} &> |\text{round}^{\mathcal{Q}}_{\infty}(x)| - |x| &\geq 0, \\ -\varepsilon^{\mathcal{Q}} &< \text{round}^{\mathcal{Q}}_{-\infty}(x) - x &\leq 0, \\ \varepsilon^{\mathcal{Q}} &> \text{round}^{\mathcal{Q}}_{+\infty}(x) - x &\geq 0, \\ -\varepsilon^{\mathcal{Q}}/2 &\leq |\text{round}^{\mathcal{Q}}_{\mathfrak{n}}(x) - x| &\leq \varepsilon^{\mathcal{Q}}/2 . \end{aligned}$$

The nearest modes are distinguished by the choice of rounding $\varepsilon^{\mathcal{Q}}/2$ up or off. The usual approach is either to constantly round up ($n\!\uparrow$) or to round to even ($n2$), such that for uniformly distributed numbers 50% of the $\varepsilon^{\mathcal{Q}}/2$ cases would be rounded up and the other 50% rounded off. Rounding to even has some advantages when considering consecutive rounding, because it prevents the so called drift [Goldberg, 1991]. Constant off-rounding ($n\!\downarrow$), however, has in some cases the great advantage that much fewer resources are needed for an associated multiplication with rounding (Section 4.2.3.1 on page 161). For the general error analysis, however, the choice of the tie breaking strategy has little effect and most of the time we will refer to the nearest rounding modes simultaneously by the index $\mathfrak{n}$.

Even without a good global relative error bound (Eq. 2.21 on the preceding page), sometimes it is advantageous to consider the relative error relation:

(2.23)
$$\text{round}^{\mathcal{Q}}_{\mu}(x) \;=\; x\,\lambda_{\mu}(x) ,$$

with $\lambda_{\mu}(x) \in \mathbb{R}^{+}_{0}$ and the bounds

(2.24)
$$\begin{aligned} \lambda_{0}(x) &\in [0, 1], \\ \lambda_{\infty}(x) &\in [1, \infty), \\ \lambda_{\mathfrak{n}}(x) &\in [0, 2] . \end{aligned}$$

The first two bounds are clear from the definition. In case of nearest rounding, $\frac{\text{round}^{\mathcal{Q}}_{\mu}(x)}{x}$ attains a maximum for the smallest x which is rounded up, i.e: $\frac{\text{round}^{\mathcal{Q}}_{\mu}(\varepsilon^{\mathcal{Q}}/2)}{\varepsilon^{\mathcal{Q}}/2} = \frac{\varepsilon^{\mathcal{Q}}}{\varepsilon^{\mathcal{Q}}/2} = 2$. We obtain the interval $[0, 2)$ if we choose to round down $\varepsilon^{\mathcal{Q}}/2$, but this is a negligible improvement. For the rounding modes $\{-\infty, +\infty\}$ there are no bounds.

The hardware efficiency of the fixed point computations has its price when it comes to roundoff errors. We review three major problems on the basis of the $\mathcal{Q}^{\text{FX}}(2, 9, 2^{-9})$ quantization with the ($n\!\uparrow$) rounding mode:

- Underflow in multiplications $2^{-5} \cdot 2^{-5} =_{\mathcal{Q}} 0$.
  This error occurs even though $2^{-5} = 1/32 = 0.03125$ is fairly large in comparison to $\varepsilon^{\mathcal{Q}} = 0.001953125$. Such underflow in particular violates the distributive law:
  $$\sum_i (a \cdot b_i) \quad \ll_{\mathcal{Q}} \quad a \cdot \sum_i b_i$$
  for positive b_i and sufficiently small a. As we have seen above this can happen very easily and the length of the sum further amplifies this effect.
- Overflow or saturation in additions $(-0.5) + (-0.7) =_{\mathcal{Q}} -1$.
  Here we have applied the saturation, which means that if the computed number would exceed the represented number range the result clamps to the last representable number with the same sign. This is especially critical in scalar products and matrix vector operations. To avoid overflow or saturation we may be forced to rescale sums $\sum_{i \in I} b_i$ by the number of addends $|I|$ and compute $\sum_{i \in I} \frac{b_i}{|I|}$ instead, however this reproduces the first problem.
- Aliasing in the evaluation of non-linear functions $f(x) := \frac{1}{512x}$, $f(\{\frac{342}{512}, \ldots, \frac{511}{512}\}) = \frac{1}{512}$.
  One third of the positive number range is mapped onto the same result. The information carried by the different values is lost and their differences cannot have any impact on the solution any more. Instead, the change from $f(\frac{342}{512}) = \frac{1}{512}$ to $f(\frac{341}{512}) = \frac{2}{512}$ artificially obtains high relevance for the solution of the algorithm although the derivative of the function at this point is very small $f'(\frac{341.5}{512}) \approx -0.0044$.

Since these problems are inherent to the fixed point format, we will have to cope with them in the algorithms. The primary and most effective strategy will be to reformulate the discrete schemes such that the whole represented number range is used and small and large factors equalize each other after factorization. On the implementational level we will also consider changing the order of operations, fixed point sensitive approximations to non-linear functions and if possible *variable precision arithmetic* in intermediate computations.

2.2.2.3 Numerical Stability

There are several ways of measuring a numerical error for a given quantization. Let us denote by $f : X \to Y$ the exact algorithm which produces an exact result $y \in Y$ for an exact input $x \in X$ and let $f^{\mathcal{Q}} : X \to Y$ denote the approximative algorithm working with some finite precision $\mathcal{Q}$. Then we may define the ***forward errors*** for $x \in X$ as

$$\begin{array}{rl} \text{absolute forward error} & |f(x) - f^{\mathcal{Q}}(x)|, \\ \text{relative forward error} & |f(x) - f^{\mathcal{Q}}(x)|/|f(x)|\,. \end{array}$$

With these errors we measure how much the approximate solution $f^{\mathcal{Q}}(x)$ obtained with the precision $\mathcal{Q}$ differs from the exact solution $f(x)$.

The ***backward errors*** for $x \in X$ are defined as

$$\begin{array}{rc} \text{absolute backward error} & \inf\limits_{\{\Delta x \in X - x \,\big|\, f(x+\Delta x) = f^{\mathcal{Q}}(x)\}} |\Delta x|, \\ \text{relative backward error} & \inf\limits_{\{\Delta x \in X - x \,\big|\, f(x+\Delta x) = f^{\mathcal{Q}}(x)\}} |\Delta x| / |x| \,. \end{array}$$

With these errors we measure by which perturbation of the data $x + \Delta x$ we can explain the approximate solution $f^{\mathcal{Q}}(x)$ we have obtained.

One can also define ***mixed forward-backward errors*** requiring that there should exist absolutely or relatively small $\Delta x, \Delta y$ such that

$$f(x + \Delta x) \;=\; f^{\mathcal{Q}}(x) + \Delta y \,.$$

Then if $\frac{|\Delta x|}{|x|}$ and $\frac{|\Delta y|}{|f(x)|}$ are small in comparison to the number of operations necessary for the evaluation of $f^{\mathcal{Q}}$ we can call the implementation $f^{\mathcal{Q}}$ ***numerically stable***. Depending on the problem one may want to vary the meaning of ”small”. A typical condition would require

$$\begin{aligned} \frac{|\Delta x|}{|x|} &\leq N^{f^{\mathcal{Q}}} u^{\mathcal{Q}} + o(u^{\mathcal{Q}}) \,, \\ \frac{|\Delta y|}{|f(x)|} &\leq N^{f^{\mathcal{Q}}} K_{\text{rel}}(f) u^{\mathcal{Q}} + o(u^{\mathcal{Q}}) \,, \end{aligned}$$

where $N^{f^{\mathcal{Q}}}$ is the number of elementary operations in $f^{\mathcal{Q}}$, $u^{\mathcal{Q}}$ is the unit roundoff and $K_{\text{rel}}(f)$ the relative condition of f defined by

$$K_{\text{rel}}(f) \;:=\; \inf\left\{ K \in \mathbb{R} \middle| \forall x : \frac{|f(\tilde{x}) - f(x)|}{|f(x)|} \leq K \frac{|\tilde{x} - x|}{|x|} + o\left(\frac{|\tilde{x} - x|}{|x|} \right) \text{ for } \tilde{x} \to x \right\}.$$

The problem with these conditions is, that they are only useful for quantizations which satisfy $u^{\mathcal{Q}} \ll 1/N^{f^{\mathcal{Q}}}$, otherwise it is hardly sensible to talk of numerical stability although Δx and x are of approximately the same order. The common *floating point* formats satisfy this condition

$$\begin{array}{rl} \text{IEEE single s23e8} & u^{\mathcal{Q}} \approx 6.0 \cdot 10^{-8} \,, \\ \text{IEEE double s52e11} & u^{\mathcal{Q}} \approx 1.1 \cdot 10^{-16} \,, \\ \text{IEEE extended s63e15} & u^{\mathcal{Q}} \approx 5.4 \cdot 10^{-20} \,, \end{array}$$

although the single precision format may already run into problems since current microprocessors execute elementary operations at several GHz = 10^9Hz.

In case of *fixed point number* formats used in graphics hardware and reconfigurable computing this definition of numerical stability makes no sense, since we have seen that $u^{\mathcal{Q}} > \frac{1}{3}$ (Eq. 2.21 on page 31). The fixed point format rather controls the absolute error such that we are better advised to require that

$$|\Delta x| \;\leq\; N^{f^{\mathcal{Q}}} \varepsilon^{\mathcal{Q}} + o(\varepsilon^{\mathcal{Q}}) \,,$$

$$|\Delta y| \quad \leq \quad N^{f^{\mathcal{Q}}} K_{\text{abs}}(f)\varepsilon^{\mathcal{Q}} + o(\varepsilon^{\mathcal{Q}})\,,$$

where $N^{f^{\mathcal{Q}}}$ is again the number of operations in $f^{\mathcal{Q}}$, $\varepsilon^{\mathcal{Q}}$ is the machine epsilon and $K_{\text{abs}}(f)$ the absolute condition of f defined by

$$K_{\text{abs}}(f) \quad := \quad \inf\left\{K \in \mathbb{R} \middle| \forall x : |f(\tilde{x}) - f(x)| \leq K|\tilde{x} - x| + o\left(|\tilde{x} - x|\right) \text{ for } \tilde{x} \to x\right\}.$$

In this case the size of the machine epsilon of the number formats available for implementation in comparison to the number of involved operations is crucial:

$$\begin{aligned} \mathcal{Q}^{\text{FX}}(2, 8, 2^{-8}) &\qquad \varepsilon^{\mathcal{Q}} \approx 3.9 \cdot 10^{-3}\,, \\ \mathcal{Q}^{\text{FX}}(2, 12, 2^{-12}) &\qquad \varepsilon^{\mathcal{Q}} \approx 2.4 \cdot 10^{-4}\,, \\ \mathcal{Q}^{\text{FX}}(2, 16, 2^{-16}) &\qquad \varepsilon^{\mathcal{Q}} \approx 1.5 \cdot 10^{-5}\,, \end{aligned} \tag{2.25}$$

where $\mathcal{Q}^{\text{FX}}(\beta, t, \varepsilon)$ has been defined in Eq. 2.20 on page 31. For a typical image processing application with tens of local operations and tens to hundreds of iterations we see that we are at loss with the signed 9 bit format. The additional 8 bit in the signed 17 bit format can make a big difference here, possibly bounding the overall error below few percent of the number range. However, even these few percent would become visually disturbing, if they formed a pattern. Therefore, in fixed point arithmetic a different approach is often chosen. Instead of trying to control the individual errors one investigates the distribution of the roundoff error and tries to show that under certain conditions, this distribution may be assumed uniform. This assumption is referred to as the ***white-noise model*** [Jackson, 1970; Urabe, 1968; Welch, 1969]. Although one can formulate conditions for which the white-noise assumption is justified [Barnes et al., 1985; Wong, 199O], the same papers demonstrate that various distributions of the input signal violate these conditions and that then the output error differs significantly from the predictions of the white-noise model. Especially when the errors are examined for digital filters and not single multiplications [Mulcahy, 1980; Wong, 1991], the error spectra differ dramatically from those derived by the white-noise model. Moreover, the output errors are also very sensitive to the characteristics of the input signal, such that it is very hard to give any reasonable bounds on the error unless the distribution of the input signals hardly varies and is known quite exactly in advance.

Recently a thorough justification of the white-noise model for *fixed point* roundoff errors in digital systems under very general conditions on the input signal and the involved coefficients has been given [Vladimirov and Diamond, 2002]. The result states that the joint probabilistic distribution of the input and the error weakly converges to the joint distribution of the input and a uniformly distributed vector as the step size of the quantization converges to zero. From a general point of view this result is very satisfactory, since it shows that the deterministic dependence of the error on the input signal breaks down in favor of a uniform distribution in the limit. However, in our implementational cases where $\varepsilon^{\mathcal{Q}}$ is fairly large we cannot deduce any bounds on the error from the asymptotic behavior.

This knotty situation, that in general neither good deterministic nor probabilistic error bounds can be attained for low precision fixed point computations, has lead to the development of fixed point error simulators and analysis tools which give the engineer the possibility to concentrate on the concrete implementation and optimize it in terms of performance and accuracy [Bauer and Leclerc, 1991; Coster et al., 1996; Kim et al., 1998; Sung and Kum, 1995; Wadekar and Parker, 1998; Willems et al., 1997a]. In conjunction with these tools, floating point to fixed point translators are used to minimize design time and to automatize number range scaling and word-length optimizations [Cmar et al., 1999; Keding et al., 1998; Willems et al., 1997b].

We will also deal individually with the different applications, but our focus lies on the preservation of global invariants and reproduction of the qualitative behavior known from the continuous models rather than to optimize the accuracy of each individual operation. This approach is most suitable for image processing applications since we are interested in certain characteristics of the image evolution and not the individual values. In this way we will be able to obtain satisfactory results despite the inability to control the component-wise deterministic or probabilistic error in the long run.

2.2.3 Quantized Operators

Here we define the quantized operators which will be used throughout the rest of the work. We analyze the possibilities of controlling global features of vector operands which undergo quantized operations. The quantization model is tailored to the *fixed point number* system, but the *floating point number* system also fulfills the same conditions, if we restrict the exponent to one or just a few scales.

In Section 2.2.2.2 on page 30 we have seen that fixed point number formats can vary in a number of parameters and in fact different hardware architectures use different formats. In graphics hardware even incompatible number formats are used simultaneously. By introducing a formal set of quantized operators we want to abstract from the low level differences and allow a common analysis of their properties.

2.2.3.1 Arithmetic Operations

Given a *quantization* $\mathcal{Q}$ we will generally assume:

- Zero element $0 \in \mathcal{Q}$.
  This is reasonable since a missing neutral element of addition causes many problems, but not all quantizations in practice fulfill this requirement. The first realization of negative numbers in graphics hardware used the unsigned set $0, \dots, 255$ and interpreted the numbers as signed numbers $-1 = \frac{2\cdot(0)-255}{255}, \dots, \frac{2\cdot(n)-255}{255}, \dots, \frac{2\cdot(255)-255}{255} = 1$. Because the numerator is always odd, there is no representation of 0 in this quantization.
- Symmetry $\quad -a \in \mathcal{Q} \;$ if $\; a \in \mathcal{Q}, a > 0$.
  This guarantees that $\mathcal{Q}$ contains a symmetric set of positive and negative numbers around zero. However, there may be more negative than positive numbers, such that some negative numbers cannot be negated in $\mathcal{Q}$.

Ideally $\mathcal{Q}$ will cover the interval $[-1, 1]$ with an exact representation of $0, 1, -1$, but in general we can only assume the above two properties.

Equivalent to the standard arithmetic operations $\circ \in \{+, -, \cdot, /\}$ we introduce ***quantized operations*** $\circledcirc \in \{\oplus, \ominus, \odot, \oslash\}$ on $\mathcal{Q}$, the notation is borrowed from [Goldberg, 1991]. Unfortunately $\mathcal{Q}$ together with the quantized operations does not constitute any of the standard algebraic objects, but some of the properties are preserved.

- ***Saturation*** $\text{clamp}^{\mathcal{Q}}(x) := \min(\max(x, \min\mathcal{Q}), \max\mathcal{Q}), \quad x \in \mathbb{R}$.
  We will assume that all operations saturate rather than overflow. In low precision hardware this functionality is often explicitly implemented, otherwise it costs little additional resources. The advantage of saturation is its continuity and monotonicity, while an overflow violates both, e.g.:

$$\begin{array}{lcc} \text{saturation} & \max\mathcal{Q} + \text{inc}^{\mathcal{Q}}(0) =_{\mathcal{Q}} \max\mathcal{Q} & \frac{511}{512} + \frac{1}{512} =_{\mathcal{Q}} \frac{511}{512}\,, \\ \text{overflow} & \max\mathcal{Q} + \text{inc}^{\mathcal{Q}}(0) =_{\mathcal{Q}} \min\mathcal{Q} & \frac{511}{512} + \frac{1}{512} =_{\mathcal{Q}} -1\,, \end{array}$$

  where $\text{inc}^{\mathcal{Q}}(0)$ is the smallest positive non-zero number and a $\mathcal{Q}^{\text{FX}}(2, 9, 2^{-9})$ format is used in the examples on the right.

- Addition $a \oplus b, \quad a, b \in \mathcal{Q}$.

$$a \oplus b \;=\; \text{clamp}^{\mathcal{Q}}(a + b)\,.$$

  This means that addition is exact as long as the result stays within the representable range, otherwise a saturation occurs.

- Subtraction $a \ominus b, \quad a, b \in \mathcal{Q}$.

$$a \ominus b \;=\; \text{clamp}^{\mathcal{Q}}(a - b)\,.$$

  Here we have a similar restriction as for the addition. We may define negation as

$$\ominus b \;:=\; 0 \ominus b\,,$$

  and then

$$a \ominus b \;=\; a \oplus (\ominus b)$$

  holds true if $-b \in \mathcal{Q}$.

- Multiplication $a \odot b, \quad a, b \in \mathcal{Q}$.

$$a \odot b \;=\; \text{round}^{\mathcal{Q}}_{\mu}(\text{clamp}^{\mathcal{Q}}(a \cdot b))\,.$$

  The *rounding modes* indicated by μ are defined in Eq. 2.22 on page 32. Since different hardware architectures have different rounding modes and we are not always free to choose among them, some results will depend on the modes. But independent of the rounding mode we have the following properties:

$$\begin{array}{rcll} 1 \odot b &=& b & \text{if } 1 \in \mathcal{Q}, \\ 0 \odot b &=& 0 & \text{if } 0 \in \mathcal{Q}, \\ \ominus a \odot b &=& \ominus(a \odot b) & \text{if } -a, -(a \odot b) \in \mathcal{Q}\,. \end{array}$$

- Division $a \oslash b, \quad a, b \in \mathcal{Q}$.

$$a \oslash b \quad = \quad \text{round}_\mu^\mathcal{Q}(\text{clamp}^\mathcal{Q}(a/b))\,.$$

We have included the division for the sake of completeness. In concrete implementations we avoid divisions by non-powers of 2, because either they are impossible to implement in the architecture or their use would cost unreasonably many resources. Where divisions are unavoidable and $|b|$ is well bounded from below, a discrete inverse function $f^\mathcal{Q}(b) := \text{round}_\mu^\mathcal{Q}(1/b)$ can be used, otherwise a two-dimensional discrete function $f^\mathcal{Q}(a,b) := \text{round}_\mu^\mathcal{Q}(a/b)$ implements the division. However, in the latter case we must also avoid a large co-domain, which would lead to frequent saturation.

2.2.3.2 Composed Operations

Once we have defined the quantized arithmetic operations, quantized composed operations such as a matrix vector product

$$\left(A \odot \bar{V}\right)_i \quad := \quad \bigoplus_j A_{ij} \odot \bar{V}_j$$

can be derived from the standard arithmetic operations in a natural way. As the formulas may be more complicated than that and in most cases all operations will be replaced by the quantized equivalents, we define a quantized ***evaluation operator*** recursively by

$$\begin{array}{rcl} \text{eval}^\mathcal{Q}(A) &:=& A, \quad A \in \mathcal{Q}^{m\times n}, \\ \text{eval}^\mathcal{Q}(\text{term1} \circ \text{term2}) &:=& \text{eval}^\mathcal{Q}(\text{term1}) \circledcirc \text{eval}^\mathcal{Q}(\text{term2})\,, \end{array}$$

where $\circ \in \{+, -, \cdot, /\}$ is the exact and $\circledcirc \in \{\oplus, \ominus, \odot, \oslash\}$ the corresponding quantized operation. Since the quantized operations are not associative in general, the result of the quantized evaluation operator depends on the order of the term splitting. We assume an evaluation order from left to right unless otherwise stated. The quantized matrix vector product now reads:

$$\text{eval}^\mathcal{Q}(A \cdot \bar{V}) \quad = \quad A \odot \bar{V}\,.$$

In all our implementations the matrix A will be a thin band matrix, such that each component sum will have only few (e.g. 9) multiplications and additions. However, overflow in additions and underflow in multiplications can occur very quickly in low precision fixed point arithmetic (see examples on page 32 in Section 2.2.2.2). Moreover, we have also seen that the order of operations may have a great impact on the result and because the matrix vector product is a key ingredient of the discrete schemes we must at least control the error distribution in its result even if the precision is too low to guarantee certain error bounds in the long run.

2.2.3.3 Matrix Vector Product

In concrete implementations one notices that despite elaborate algorithm adaption to the *fixed point number* system, the restricted precision still causes a significant deviation from the continuous results. But in the presented PDE based applications in image processing we are not interested in the exact continuous process, at least not practically, but in an image evolution which is governed by the characteristics modeled through the PDE. This qualitative behavior can be reconstructed with a quantized solver, if we are able to control global attributes despite arbitrary large local errors due to a coarse quantization.

The sum of all elements of a vector $\bar{V}$, representing the overall mass in the problem, is such a global attribute. The elements of $\bar{V}$ are typically changed by the application of a matrix, say A. Formally the mass is preserved if A has unit column sum, as $\sum_i A_{ij} = 1$ for all j implies:

$$\sum_i \left(A\bar{V}\right)_i = \sum_i \sum_j A_{ij}\bar{V}_j \;=\; \sum_j \bar{V}_j \sum_i A_{ij} = \sum_j \bar{V}_j. \tag{2.26}$$

Considering the underflow in multiplications we have seen in the previous Section 2.2.2 on page 26, it is clear that we cannot preserve the equality in this formulation. For arbitrary numbers even *floating point* calculations violate it. Special ***exact dot product*** operations, which perform an exact computation of a scalar product of two vectors with a single final rounding after the summation, have been proposed to reduce roundoff in such matrix vector multiplications. Although in this way iterative applications of A can still accumulate errors, the error is not larger than produced by a series of basic arithmetic operations. At least for standard floating point number formats this usually means that the error becomes negligible, but multiple iterations remain an issue for low precision fixed point formats even with the exact dot product. For more details on this and related variable precision techniques we refer to [Schulte and Swartzlander, 1996]. We will use *variable precision arithmetic* in intermediate computations, when our hardware allows the required configuration (Section 4.2.2.2 on page 159). For inexpensive standard hardware products (e.g. graphics hardware in Section 4.1 on page 110), which have processing elements of fixed width, we must come up with a different solution.

If A is symmetric and has unit row sum we may rewrite the matrix vector product as

$$\begin{aligned} A\bar{V} &= \bar{V} + (A-I)\bar{V} \\ \left((A-I)\bar{V}\right)_i &= \sum_j A_{ij}\bar{V}_j - 1\cdot\bar{V}_i = \sum_j A_{ij}\bar{V}_j - \sum_j A_{ij}\cdot\bar{V}_i = \sum_{j:j\neq i} A_{ij}(\bar{V}_j - \bar{V}_i) \\ \left(A\bar{V}\right)_i &= \bar{V}_i + \sum_{j:j\neq i} A_{ij}(\bar{V}_j - \bar{V}_i)\,. \end{aligned} \tag{2.27}$$

In this way we have encoded the constraint of unit row sum implicitly in the formula, i.e. we do not use A_{ii} in the evaluation any more, but the formula implies that $A_{ii} = 1 - \sum_{j:j\neq i} A_{ij}$. The advantage we gain is the anti-symmetry of the addends $F_{ij} := A_{ij}(\bar{V}_j - \bar{V}_i)$, i.e. what F_{ij} contributes to $\left(A\bar{V}\right)_i$, the negative $-F_{ij}$ contributes to $\left(A\bar{V}\right)_j$. For the mass preservation it therefore does not matter how inexact we have computed F_{ij}. This pattern is well known from Finite Volume discretizations, where one computes anti-symmetric fluxes across cell boundaries and the addends F_{ij} are fluxes in this sense, but here they are associated with interactions between arbitrary nodes and not only cell neighbors. So the scheme can be applied to any type of discretization as long as the resulting matrix is symmetric with unit row sum.

The formula in Eq. 2.27 on the preceding page actually applies to arbitrary symmetric matrices with zero row sum like $Z^A := (A - I)$, because $Z^A_{ij} = A_{ij}$ for all i, j with $j \neq i$:

$$(2.28) \qquad \left(Z^A \bar{V}\right)_i \;=\; \sum_{j:j\neq i} Z^A_{ij}(\bar{V}_j - \bar{V}_i)\,.$$

If we perform exact additions and subtractions it is guaranteed that the element sum of the result vector $Z^A\bar{V}$ is zero. An arbitrary symmetric matrix A can be split uniquely

$$A = Z^A + S^A$$

into a symmetric zero row sum matrix Z^A and a diagonal matrix S^A. The decomposition is unique because the elements of S^A must contain the row sums of A to make Z^A a zero row sum matrix. We obtain

$$\begin{aligned} S^A_{ii} &= \sum_j A_{ij}, \\ Z^A_{ij} &= A_{ij}, \qquad j \neq i, \\ Z^A_{ii} &= -\sum_{j:j\neq i} A_{ij}\,. \end{aligned}$$

Then we can apply Z^A to produce a zero mass result vector, while the product with S^A will assign the new mass to the result. We define this ***mass-exact matrix vector product*** in the quantized setting:

$$(2.29) \qquad \begin{aligned} (A \boxdot \bar{V})_i \;&:=\; \mathrm{eval}^{\mathcal{Q}}\left(S^A_{ii}\bar{V}_i \;+\; \sum_{j:j\neq i} A_{ij}(\bar{V}_j - \bar{V}_i)\right) \\ &=\; \left(S^A_{ii} \odot \bar{V}_i\right) \;\oplus\; \bigoplus_{j:j\neq i} A_{ij} \odot (\bar{V}_j \ominus \bar{V}_i)\,. \end{aligned}$$

In future we will use the $\boxdot$ subscript for the $\mathrm{eval}^{\mathcal{Q}}$ quantized evaluation operator to indicate that the term evaluation makes use of this mass-exact matrix vector product, i.e:

$$\begin{aligned} \mathrm{eval}^{\mathcal{Q}}_{\boxdot}(A\bar{V}) &= (A \boxdot \bar{V})\,, \\ \mathrm{eval}^{\mathcal{Q}}(A\bar{V}) &= (A \odot \bar{V})\,. \end{aligned}$$

If no saturation in the evaluation of $A \boxdot \bar{V}$ occurs, then this matrix vector product has the property

$$\sum_i (A \boxdot \bar{V})_i \;=\; \sum_i (S^A \odot \bar{V})_i\,,$$

and in the special case of a unit row sum matrix $A = Z^A + I$, the mass is preserved:

$$\sum_i (A \boxdot \bar{V})_i \;=\; \sum_i \bar{V}_i \,. \tag{2.30}$$

2.2.3.4 Scaling

We notice that the use of the differences $\bar{V}_j - \bar{V}_i$ in Eq. 2.29 on the preceding page may easily lead to an underflow in the subsequent multiplication, because we will have $A_{ij} \neq 0$ exclusively for small distances $\|i - j\|$, i.e. for nodes lying closely together, and these tend to have similar values. So especially in smooth image areas we should expect a high error. To increase accuracy we can rescale the formula

$$(A \boxdot_s \bar{V})_i \;:=\; \left(S^A_{ii} \odot \bar{V}_i\right) \oplus \frac{1}{s_i} \odot \bigoplus_{j:j\neq i} A_{ij} \odot s_i \odot (\bar{V}_j \ominus \bar{V}_i)\,, \tag{2.31}$$

with some scaling factors s_i as long as no saturation occurs. For integer s_i and small $(\bar{V}_j \ominus \bar{V}_i)$ the scaling $s_i \odot (\bar{V}_j \ominus \bar{V}_i) = s_i(\bar{V}_j - \bar{V}_i)$ is exact and will be performed first. The absolute forward error then decreases:

$$\begin{aligned}
\left|(A \boxdot \bar{V})_i - (A\bar{V})_i\right| &\leq \frac{\varepsilon^{\mathcal{Q}}}{1+\delta_{\mu\in\mathfrak{n}}}\left(1 + \#\{j \big| A_{ij} \neq 0\}\right)\,,\\
\left|(A \boxdot_s \bar{V})_i - (A\bar{V})_i\right| &\leq \frac{\varepsilon^{\mathcal{Q}}}{1+\delta_{\mu\in\mathfrak{n}}}\left(2 + \frac{\#\{j \big| A_{ij} \neq 0\}}{s_i}\right)\,,
\end{aligned}$$

where μ indicates the dependence on the rounding mode (see Eq. 2.22 on page 32). Certainly this only makes sense if $s_i > 1$, i.e. the image in the area $\{j \big| A_{ij} \neq 0\}$ is sufficiently smooth. This should be the case in most parts of an image, however, in the vicinity of sharp edges $s_i \leq 1$ is almost inevitable. Ultimately the choice of s_i depends on the matrix elements $\{j \big| A_{ij} \neq 0\}$, since we must prevent individual addends and the whole sum $\bigoplus_{j:j\neq i} A_{ij} \odot s_i \odot (\bar{V}_j \ominus \bar{V}_i)$ from saturation. For large elements of A_{ij} we would even have to apply the contrary strategy and scale down the addends with an $s_i < 1$ before addition and scale up the result with $\frac{1}{s_i} > 1$ afterwards. In this case the scaling with s_i would be applied implicitly by including the factor into the matrix assembly of the elements $\{j \big| A_{ij} \neq 0\}$, such that the multiplication $A_{ij} \odot s_i = A_{ij} \cdot s_i$ would be exact.

The greatest disadvantage of the scaling is that in general we lose the property of the mass-exact computation. Even if we chose $\forall i : s_i := s$ to be a global scaling factor, such that the addends would remain antisymmetric, the multiplication of the sum with $\frac{1}{s}$ would round-off differently depending on the value of the sum. Only in the case of scaling down the addends to prevent saturation we can retain the mass-exact computation by choosing s_i as the inverse of an integer, such that $\frac{1}{s_i}$ is an integer and the rescaling of the sum with this factor is exact. This means that we can always prevent the sum in Eq. 2.29 on the preceding page from saturation and thus preserve the mass-exact computation, but this may lead to an increase in the absolute error. Otherwise, if we sacrifice the exact mass property we can substantially decrease the error in smooth image areas.

We could also apply the scaling directly to the natural matrix vector product:

$$(2.32) \qquad \left(A \odot_s \bar{V}\right)_i \;:=\; \left(S^A_{ii} \odot \bar{V}_i\right) \,\oplus\, \frac{1}{s_i} \odot \bigoplus_j A_{ij} \odot s_i \odot \bar{V}_j \,.$$

There are two differences to the scaled version of the mass-exact matrix vector product as in Eq. 2.31 on the preceding page. First, in Eq. 2.31 some of the roundoff errors will still be compensated by the partial antisymmetry of the addends especially if we use a global scaling factor $\forall i : s_i := s$, such that the mass defect should turn out smaller than in Eq. 2.32. Second, the scaling in Eq. 2.31 is independent of the absolute intensities and uses only their differences, such that it will have the same impact on each level of intensities, while Eq. 2.32 will only rescale small values. It is therefore not advisable to use a global scaling factor in this formula.

In practice, we will usually not need to scale down the addends to prevent a saturation because the A_{ij} will be sufficiently small. Although this will increase the risk of underflow in multiplications with small differences we will abstain from scaling up the addends because the accurate processing of smooth areas is usually less important than the accurate treatment of edges. Moreover, the global property of preserving the mass in case of unit row sum in A produces higher quality results than uncorrelated high local accuracy.

2.2.3.5 Backward Error

Let us consider the effects of the roundoff errors in the mass-exact matrix vector product as a perturbation of the matrix. If we express the roundoff error in multiplications in the multiplicative form (Eq. 2.23 on page 32) and assume no overflow in additions then we have for a symmetric matrix A

$$\begin{aligned}
(A \boxdot \bar{V})_i &= \left(S^A_{ii} \odot \bar{V}_i\right) \,\oplus\, \bigoplus_{j:j\neq i} A_{ij} \odot (\bar{V}_j \ominus \bar{V}_i) \\
&= \lambda^{ii}_\mu S^A_{ii} \bar{V}_i \,+\, \sum_{j:j\neq i} \lambda^{ij}_\mu A_{ij} (\bar{V}_j - \bar{V}_i) \\
&= \left(\lambda^{ii}_\mu S^A_{ii} - \sum_{j:j\neq i} \lambda^{ij}_\mu A_{ij}\right) \bar{V}_i \,+\, \sum_{j:j\neq i} \lambda^{ij}_\mu A_{ij} \bar{V}_j \,,
\end{aligned}$$

with the abbreviations $\lambda^{ij}_\mu := \lambda_\mu(A_{ij}(\bar{V}_j - \bar{V}_i))$ and $\lambda^{ii}_\mu := \lambda_\mu(S^A_{ii}\bar{V}_i)$. The last formula line has the form of a natural matrix vector product. This means that we have

$$(2.33) \qquad A \boxdot \bar{V} \;=\; B^{A,\bar{V}} \cdot \bar{V}$$

with the matrix

$$(2.34) \qquad (B^{A,\bar{V}})_{ij} \;:=\; \lambda^{ij}_\mu A_{ij}, \qquad j \neq i,$$

$$(B^{A,\bar{V}})_{ii} \quad := \quad \lambda^{ii}_{\mu} S^{A}_{ii} - \sum_{j:j\neq i} \lambda^{ij}_{\mu} A_{ij} \,.$$

Thus we can interpret the roundoff error as a component-wise absolute backward error from $B^{A,\bar{V}}$ to A:

$$\begin{aligned} |\lambda^{ij}_{\mu} A_{ij} - A_{ij}| &\leq \frac{\varepsilon^{\mathcal{Q}}}{1+\delta_{\mu\in\mathfrak{n}}} \cdot \frac{1}{|\bar{V}_j - \bar{V}_i|}, \qquad j \neq i, \\ \left| \lambda^{ii}_{\mu} S^{A}_{ii} - \sum_{j:j\neq i} \lambda^{ij}_{\mu} A_{ij} - A_{ii} \right| &\leq \frac{\varepsilon^{\mathcal{Q}}}{1+\delta_{\mu\in\mathfrak{n}}} \left(\frac{1}{|\bar{V}_i|} + \sum_{\substack{j:j\neq i \\ A_{ij}\neq 0}} \frac{1}{|\bar{V}_j - \bar{V}_i|} \right), \end{aligned}$$

where the term $\frac{1}{|\bar{V}_i|}$ disappears if A has integer row sum. We see again that the scheme produces large errors in areas where the data is smooth and is fairly precise in areas of high gradients. The bound for the diagonal elements looks catastrophic because all errors from the sub-diagonals are subtracted here and their interference is unknown. In practice is it highly improbable that all roundoffs will point in the same direction, such that usually error compensation will occur. But we cannot quantify this effect, since the errors are correlated and the white-noise model fails here. In all iterative smoothing processes, however, we have a self regulated accuracy, since any exceptionally high random errors in smooth areas lead to an increase in differences and thus to a more precise smoothing of this area in the next step.

2.3 Anisotropic Diffusion

We recall the problem statement from Section 2.1.1 on page 15:

$$\begin{aligned} \partial_t u - \operatorname{div}\left(G(\nabla u_\sigma)\nabla u\right) &= 0 \;, \quad \text{in } \mathbb{R}^+ \times \Omega \,, \\ u(0) &= u_0 \,, \quad \text{on } \Omega \,, \\ G(\nabla u_\sigma)\nabla u \cdot \nu &= 0 \;, \quad \text{on } \mathbb{R}^+ \times \partial\Omega \,. \end{aligned}$$

The non-linear diffusion coefficient $G(\nabla u_\sigma)$ steers the diffusion process in such a way that edges are preserved while noise in homogenous areas is smoothed.

2.3.1 FE Scheme

Finite Element (FE) methods are widely used to discretize this partial differential equation and were for instance studied by Kačur and Mikula [Kačur and Mikula, 1995]. Weickert [Weickert, 1998] proposed additive operator splitting techniques based on finite difference schemes to accelerate the solution process. Multi-grid methods have also been considered [Acton, 1998], which allow for a better convergence as long as the anisotropic behavior of the diffusion process is not too strong. The use of adaptive FEs has been discussed in [Bänsch and Mikula, 1997; Preußer and Rumpf, 2000]. Moreover parallel computing has also been applied in e.g. [Weickert et al., 1997]. We will use a simple FE scheme which is well suited for parallelization and minimizes underflow in multiplications by cancellation of large and small factors.

2.3.1.1 Discrete Schemes

We use the discretization scheme described in Section 2.2.1 on page 24 with the operator $\mathcal{F}[\mathcal{C}^p_\sigma[u],u] = -\mathrm{div}\left(\mathcal{C}^p_\sigma[u]\nabla u\right)$ and the classifier $\mathcal{C}^p_\sigma[u] = G(\nabla u_\sigma)$, and either the explicit (Eq. 2.12 on page 25) or the semi-implicit (Eq. 2.14 on page 25) time-discretization. In space we discretize the problem with bilinear Finite Elements (FEs) on a uniform quadrilateral grid (Figure 2.1 on page 27). In variational formulation with respect to the FE space $\mathcal{V}^h$ we obtain for the explicit time-discretization

$$\left(U^{n+1},\Theta\right)_h = \left(U^n,\Theta\right)_h - \tau^n\left(G(\nabla U^n_\sigma)\nabla U^n,\nabla\Theta\right)$$

and for the semi-implicit time-discretization

$$\left(U^{n+1},\Theta\right)_h + \tau^n\left(G(\nabla U^n_\sigma)\nabla U^{n+1},\nabla\Theta\right) = \left(U^n,\Theta\right)_h$$

for all $\Theta \in \mathcal{V}^h$, where capital letters indicate the discrete functions in $\mathcal{V}^h$ corresponding to the continuous functions. Above $(\cdot,\cdot)$ denotes the L^2 product on the domain Ω, $(\cdot,\cdot)_h$ is the lumped mass product [Thomée, 1984], which approximates the L^2 product, and τ^n the current time-step width. The discrete solution U^n is expected to approximate $u(\sum_{i=0}^{n-1}\tau^i)$.

Thus in the n-th time-step we have to solve either the explicit

$$M^h\bar{U}^{n+1} = \left(M^h - \tau L[\bar{U}^n_\sigma]\right)\bar{U}^n_\sigma$$

or the implicit linear equation system

$$\left(M^h + \tau L[\bar{U}^n_\sigma]\right)\bar{U}^{n+1} = M^h\bar{U}^n\,,$$

where $(\bar{U}^n_\alpha)_\alpha$ is the solution vector consisting of the nodal values at the grid nodes $\alpha \in [0,N_x-1]\times[0,N_y-1]$ (Figure 2.1 on page 27). We have the lumped mass matrix

$$M^h_{\alpha\beta} := (\Phi_\alpha,\Phi_\beta)_h = h^2\mathbb{1} \tag{2.35}$$

and the weighted stiffness matrix

$$L[\bar{U}^n_\sigma]_{\alpha\beta} := \left(G(\nabla\bar{U}^n_\sigma)\nabla\Phi_\alpha,\nabla\Phi_\beta\right), \tag{2.36}$$

where Φ_α are the multi-linear basis functions. Finally we obtain the explicit update formula

$$\bar{U}^{n+1} = A^-[\bar{U}^n_\sigma]\cdot\bar{U}^n \tag{2.37}$$

$$A^-[\bar{U}^n_\sigma] \;:=\; \mathbb{1} - \frac{\tau^n}{h^2} L[\bar{U}^n_\sigma]$$

and the linear equation system for the implicit case

$$\begin{aligned} A^+[\bar{U}^n_\sigma]\cdot \bar{U}^{n+1} &= \bar{U}^n \\ A^+[\bar{U}^n_\sigma] &:= \mathbb{1} + \frac{\tau^n}{h^2} L[\bar{U}^n_\sigma]\,. \end{aligned} \tag{2.38}$$

The great advantage of the simple equidistant grid is the factorization of h^2 from M^h in Eq. 2.35 on the facing page, such that in the matrices A^- and A^+ the time-step τ^n can be balanced against the grid size square h^2, and most coefficients in $L[\bar{U}^n_\sigma]$ are of the same order and can be well represented by *fixed point numbers*.

The linear equation system is solved approximately by an iterative solver:

$$\begin{aligned} \bar{X}^0 &= \bar{R} := \bar{U}^n \\ \bar{X}^{l+1} &= F(\bar{X}^l) \\ \bar{U}^{n+1} &= F(\bar{X}^{l_{\max}})\,, \end{aligned}$$

for a fixed or adaptive number of iterations $l_{\max} \in \mathbb{N}$. Typical solvers are the Jacobi solver

$$F(\bar{X}^l) \;=\; D^{-1}(\bar{R} - (A-D)\bar{X}^l), \qquad D := \mathrm{diag}(A) \tag{2.39}$$

and the conjugate gradient solver

$$F(\bar{X}^l) \;=\; \bar{X}^l + \frac{\bar{r}^l\cdot\bar{p}^l}{A\bar{p}^l\cdot\bar{p}^l}\bar{p}^l, \qquad \bar{p}^l = \bar{r}^l + \frac{\bar{r}^l\cdot\bar{r}^l}{\bar{r}^{l-1}\cdot\bar{r}^{l-1}}\bar{p}^{l-1}, \quad \bar{r}^l = \bar{R} - A\bar{X}^l\,. \tag{2.40}$$

In the following we will see that the matrix A^+ fulfills the properties under which the above schemes converge.

2.3.1.2 Weighted Stiffness Matrix

In the general case of *anisotropic diffusion*, $\big(G(\nabla U^n_\sigma)_{i,j}\big)_{i,j\in\{x,y\}}$ is a 2 by 2 matrix. The integration in Eq. 2.36 on the preceding page is performed separately on each grid element by applying the midpoint rule to the tensor components $G(\nabla U^n_\sigma)_{i,j}$:

$$\begin{aligned} L[\bar{U}^n_\sigma]_{\alpha\beta} &= \sum_{E\in E(\alpha)}\ \sum_{i,j\in\{x,y\}} (G^n_E)_{i,j}(S^{\alpha\beta}_E)_{i,j} \\ G^n_E &:= G(\nabla U^n_\sigma(m_E)) \;\approx\; G(\nabla U^n_\sigma(x)) \qquad \forall x\in E \\ (S^{\alpha\beta}_E)_{i,j} &:= (\partial_i\Phi_\alpha, \partial_j\Phi_\beta)|_E\,, \end{aligned} \tag{2.41}$$

where $E(\alpha)$ is defined as the set of the 4 elements around the node α and m_E is the center of the element E. The vector $\nabla U^n_\sigma(m_E)$ is computed with finite differences from the neighboring nodes of U^n_σ. The result is then inserted into the transformation matrix

$$B(\nabla U^n_\sigma(m_E)) := \frac{1}{\|\nabla U^n_\sigma(m_E)\|}\begin{pmatrix} \partial_x U^n_\sigma(m_E) & \partial_y U^n_\sigma(m_E) \\ -\partial_y U^n_\sigma(m_E) & \partial_x U^n_\sigma(m_E)\end{pmatrix}, \tag{2.42}$$

and the weight matrix (Eq. 2.4 on page 17)

$$g(\|\nabla U^n_\sigma(m_E)\|) := \begin{pmatrix} g_1(\|\nabla U^n_\sigma(m_E)\|) & 0 \\ 0 & g_2(\|\nabla U^n_\sigma(m_E)\|) \end{pmatrix}. \tag{2.43}$$

With these matrices all components of the *diffusion tensor*

$$\begin{aligned} G(\nabla U^n_\sigma(m_E)) &= B(\nabla U^n_\sigma(m_E))^\top \, g(\|\nabla U^n_\sigma(m_E)\|) \, B(\nabla U^n_\sigma(m_E)) \\ &= \begin{pmatrix} g_1^E(b_x^E)^2 + g_2^E(b_y^E)^2 & (g_1^E - g_2^E)b_x^E b_y^E \\ (g_1^E - g_2^E)b_x^E b_y^E & g_1^E(b_y^E)^2 + g_2^E(b_x^E)^2 \end{pmatrix} \\ g_1^E &:= g_1(\|\nabla U^n_\sigma(m_E)\|), \quad g_2^E := g_2(\|\nabla U^n_\sigma(m_E)\|) \\ b_x^E &:= \frac{\partial_x U^n_\sigma(m_E)}{\|\nabla U^n_\sigma(m_E)\|}, \quad b_y^E := \frac{\partial_y U^n_\sigma(m_E)}{\|\nabla U^n_\sigma(m_E)\|} \end{aligned} \tag{2.44}$$

can be evaluated. The constants $(S_E^{\alpha\beta})_{i,j}$ depend only on the offset $\gamma := \beta - \alpha$ and the relative position of E to α and can be precomputed analytically. We obtain formulas for the inner sum in Eq. 2.41 on the page before

$$\sum_{i,j\in\{x,y\}} (G_E^n)_{i,j}(S_E^{\alpha\beta})_{i,j} = \begin{cases} +\left(\frac{g_1^E+g_2^E}{3} - (g_2^E - g_1^E)\frac{b_x^E b_y^E}{2}\right) & \text{if } \alpha = \beta \\ -\left(\frac{g_1^E+g_2^E}{6} - (g_2^E - g_1^E)\frac{b_x^E b_y^E}{2}\right) & \text{if } |\alpha - \beta| = (1,1) \\ -\left(\frac{g_1^E+g_2^E}{12} - (g_2^E - g_1^E)\frac{(b_x^E)^2-(b_y^E)^2}{4}\right) & \text{if } |\alpha - \beta| = (1,0) \\ -\left(\frac{g_1^E+g_2^E}{12} + (g_2^E - g_1^E)\frac{(b_x^E)^2-(b_y^E)^2}{4}\right) & \text{if } |\alpha - \beta| = (0,1) \\ 0 & \text{else} \end{cases} . \tag{2.45}$$

The anisotropic diffusion process consists of two components. The first addend in the above formula performs an isotropic linear (g_2^E) and non-linear (g_1^E) diffusion, while the second is responsible for the anisotropic behavior. The choice of the functions g_1, g_2 (Eq. 2.43) should guarantee that for all off-diagonal elements ($\alpha \neq \beta$) the results are non-positive, as this is important for the analysis later on. In general we assume $1 \geq g_2 \geq g_1 \geq 0$, then we may choose for example $g_1(x) = \frac{1}{2} + \frac{1}{2}\mathcal{P}_{c_g}(x), g_2(x) = 1$ or $g_1(x) = \frac{1}{2}\mathcal{P}_{c_g}(x), g_2(x) = \mathcal{P}_{c_g}(x)$. We could also change the weights of the isotropic and anisotropic components or use a more sophisticated approximation for $(G(\nabla \bar{U}^n_\sigma)\nabla\Phi_\alpha, \nabla\Phi_\beta)|_E$ to achieve the uniform non-positivity. Weickert proves that one can always find an appropriate approximation with finite differences for a sufficiently large stencil [Weickert, 1998, p.88f].

For the *non-linear diffusion* model when $G(\nabla \bar{U}^n_\sigma) = \tilde{g}(\nabla \bar{U}^n_\sigma)$ is a scalar we have the simpler form of the weighted stiffness matrix

$$\begin{aligned} L'[\bar{U}^n_\sigma]_{\alpha\beta} &= \sum_{E\in E(\alpha)} G_E'^n S_E'^{\alpha\beta} \\ G_E'^n &:= \tilde{g}(\nabla U^n_\sigma(m_E)) \approx \tilde{g}(\nabla U^n_\sigma(x)) \qquad \forall x \in E \end{aligned} \tag{2.46}$$

$$S_E'^{\alpha\beta} \;:=\; (\nabla\Phi_\alpha, \nabla\Phi_\beta)|_E \;=\; (S_E^{\alpha\beta})_{x,x} + (S_E^{\alpha\beta})_{y,y}\,.$$

If we set $\forall x : g_1(x) = g_2(x) = \tilde{g}(x)$ we have $G_E'^n S_E'^{\alpha\beta} = \sum_{i,j\in\{x,y\}}(G_E^n)_{i,j}(S_E^{\alpha\beta})_{i,j}$ and the formula in Eq. 2.45 on the preceding page remains valid. In particular the off-diagonal elements are non-positive for all positive choices of $\tilde{g}(x)$. Finally, in the simplest case of the *linear diffusion* we would have $\tilde{g}(x) = 1$. The mollification $\bar{U}_\sigma^n$ according to the parameter σ is performed in this way.

Let us now collect the properties of the weighted stiffness matrix $L[\bar{U}_\sigma^n]$. We have seen that $L[\bar{U}_\sigma^n]$ has non-positive off-diagonal elements if either we choose g_1, g_2 appropriately or lessen the effect of the anisotropic addend in Eq. 2.45 on the facing page. Moreover $L[\bar{U}_\sigma^n]$ has zero row sum:

$$\begin{aligned}
\sum_\beta L[\bar{U}_\sigma^n]_{\alpha\beta} &= \sum_{E\in E(\alpha)}\;\sum_{i,j\in\{x,y\}}(G_E^n)_{i,j}\left(\sum_\beta (S_E^{\alpha\beta})_{i,j}\right) = 0\\
\text{as}\quad \sum_\beta (S_E^{\alpha\beta})_{i,j} &= \Big(\partial_i\Phi_\alpha, \partial_j \underbrace{\sum_\beta \Phi_\beta}_{=1}\Big)\Big|_E = 0\,.
\end{aligned}$$

Obviously the same argument applies to $L'[\bar{U}_\sigma^n]$ and $S_E'^{\alpha\beta}$ (Eq. 2.46 on the preceding page). Under the general assumption $1 \geq g_2 \geq g_1 \geq 0$ we can also uniformly bound the diagonal elements from above using Eqs. 2.41 and 2.45 on the facing page:

$$\begin{aligned}
|L[\bar{U}_\sigma^n]_{\alpha\alpha}| &\leq 4\cdot\left|\frac{g_1^E + g_2^E}{3} - (g_2^E - g_1^E)\frac{b_x^E b_y^E}{2}\right|\\
&\leq 4\cdot\left(\frac{g_1^E + g_2^E}{3} + \frac{g_2^E - g_1^E}{4}\right)\\
&\leq 4\cdot\frac{7g_2^E + g_1^E}{12} \leq \frac{8}{3}.
\end{aligned}\tag{2.47}$$

The following properties for the matrices $A^\pm[\bar{U}_\sigma^n] = \mathbb{1} \pm \frac{\tau^n}{h^2}L[\bar{U}_\sigma^n]$ follow

- The matrices $A^\pm$ are symmetric and have unit row sum.
- All off-diagonal elements of A^- are non-negative and the diagonal is positive if $\frac{\tau^n}{h^2} < \frac{1}{\max_\alpha |L_{\alpha\alpha}|} \leq \frac{3}{8}$.
- All off-diagonal elements of A^+ are non-positive, while the diagonal is positive.
- A^+ is strictly diagonally dominant:

  $$A^+_{\alpha\alpha} = 1 - \sum_{\beta:\beta\neq\alpha} A^+_{\alpha\beta} > \sum_{\beta:\beta\neq\alpha}|A^+_{\alpha\beta}|\,,$$

  where we used the fact that A^+ has unit row sum and negative off-diagonal elements. We gather that the iterative schemes Eqs. 2.39 and 2.40 on page 45 converge for A^+.

2.3.2 Quantized Diffusion

By applying the *mass-exact matrix vector product* (Eq. 2.29 on page 40) to the discrete scheme in Eq. 2.37 on page 44 we obtain a quantized explicit mass preserving scheme

$$\begin{aligned} \bar{U}^{n+1} &= \mathrm{eval}^{\mathcal{Q}}_{\boxdot}\left(A^{-}[\bar{U}^n_\sigma]\cdot\bar{U}^n\right) \\ &= A^{-}[\bar{U}^n_\sigma]\boxdot\bar{U}^n\,, \\ &= \bar{U}^n\ominus\left(\frac{\tau^n}{h^2}L[\bar{U}^n_\sigma]\right)\boxdot\bar{U}^n\,, \end{aligned} \tag{2.48}$$

with the restriction $\frac{\tau^n}{h^2}<\frac{3}{8}$. The factor $\frac{\tau^n}{h^2}$ must be included in the matrix assembly of $L[\bar{U}^n_\sigma]$ because in the scalar vector multiplication $\frac{\tau^n}{h^2}\odot\left(L[\bar{U}^n_\sigma]\boxdot\bar{U}^n\right)$ it would be impossible to ensure that the result has zero mass even though it would be true for the input $L[\bar{U}^n_\sigma]\boxdot\bar{U}^n$.

With the semi-implicit time-discretization we eliminate the problem of restricted time-step since A^+ is strictly diagonally dominant for all τ^n, but we encounter the problem of mass inexact scalar multiplications when evaluating the iterative solvers for the linear equation system (Eq. 2.38 on page 45). Neither the scalar vector multiplications in the conjugate gradient solver (Eq. 2.40 on page 45) nor the diagonal matrix vector multiplication in the Jacobi solver (Eq. 2.39 on page 45) can be performed in a mass-exact fashion, because the local multiplicative roundoff errors do not necessarily cancel out across all elements. In case of the Jacobi solver we can minimize this effect by reformulating the scheme into a convex combination:

$$\begin{aligned} &D^{-1}(\bar{R}-(A^+-D)\bar{X}^l) \\ &= \left(\mathbb{1}+\frac{\tau^n}{h^2}L_D\right)^{-1}\left(\bar{R}-\frac{\tau^n}{h^2}(L-L_D)\bar{X}^l\right), \qquad L_D := \mathrm{diag}(L) \end{aligned}$$

$$= \Lambda\bar{R}\;+\;(\mathbb{1}-\Lambda)\left(L_D^{-1}(L_D-L)\right)\bar{X}^l, \qquad \Lambda := \left(\mathbb{1}+\frac{\tau^n}{h^2}L_D\right)^{-1} \tag{2.49}$$

$$= \bar{X}^l+\Lambda\left(\bar{R}-\bar{X}^l\right)\;+\;(\mathbb{1}-\Lambda)\left(-L_D^{-1}L\bar{X}^l\right)\,. \tag{2.50}$$

The second reformulation (Eq. 2.49) has the advantage that only positive coefficients are involved and that the convex combination favors roundoff error cancellations. Ideally the addends are evaluated by a two dimensional lookup table. Otherwise it is usually better to explicitely factor out $\bar{X}^l$ to stabilize the iterations as in the last formula (Eq. 2.50). In both cases the time-step can take arbitrary values, even those which cannot be represented, because the coefficients of Λ will be evaluated by a lookup table or a linear approximation and the results are always bounded by 1. In practice, however, the range for τ^n is also restricted similar to the explicit case, because too large τ^n will imply $\Lambda=\mathbb{1}$ in the quantized application. Therefore, implicit methods cannot play out there full strength in low precision arithmetic and despite their restrictions explicit methods are often preferable in such cases.

If we want to guarantee mass preservation for the semi-implicit scheme we must correct the mass defect after the iterations. This can be done by computing the overall mass defect

$$m_d \;:=\; \sum_\alpha\left(\bar{R}_\alpha-F(\bar{X}^{l_{\max}})_\alpha\right)\,, \tag{2.51}$$

and dithering it back to the solution

$$\bar{U}^{n+1} \;=\; F(\bar{X}^{l_{\max}}) \;+\; \text{dither}(m_d)\,, \tag{2.52}$$

where $\text{dither}(m_d)$ is a vector in which the defect m_d has been uniformly distributed across all components. Apart from keeping the mass balance, such additional dither favors the uniform distribution of roundoff errors [Wong, 1990]. However, losing the deterministic nature of the process may be undesirable, even if the random effects are marginal.

2.3.3 Quantized Scale-Space

In this section we will analyze the qualitative properties of the quantized diffusion processes in detail. This analysis is not only of direct interest for the implementations of the anisotropic diffusion methods, but also helps to understand the impact of roundoff error on other processes, since diffusion is a very common ingredient in various models.

We recall the properties of the continuous *scale-space* (Section 2.1.1.2 on page 17) and try to maintain them in the quantized setting. Similar to the continuous scale-space operator S_t we now have a discrete quantized scale-space operator

$$S_n^{\mathcal{Q}}[\bar{U}^0] \;:=\bar{U}^n\,,$$

which, given an initial vector $\bar{U}^0$ which represents an image, delivers the quantized solution $\bar{U}^n$ after n time-steps by applying the quantized explicit mass preserving scheme in Eq. 2.48 on the facing page. The following analysis transfers the results of the discrete anisotropic filtering from [Weickert, 1998, Chapter 4] to the quantized setting. The subsequent properties hold true for a vector $\bar{V}$, $\alpha \in \Omega_h$ and $n, m \in \mathbb{N}_0$:

- Semigroup property

$$\begin{aligned} S_0^{\mathcal{Q}} &= \mathbb{1}, \\ S_{n+m}^{\mathcal{Q}} &= S_n^{\mathcal{Q}} \circ S_m^{\mathcal{Q}}\,. \end{aligned}$$

  PROOF This is clear from the definition. □

- Grey level shift invariance

$$\begin{aligned} S_n^{\mathcal{Q}}[0] &= 0, \\ S_n^{\mathcal{Q}}[\bar{V} + c\bar{1}] &= S_n^{\mathcal{Q}}[\bar{V}] + c\bar{1}, \quad c \in \mathcal{Q}\,, \\ \bar{1} &:= (1, \dots, 1)^\top\,. \end{aligned}$$

  PROOF In the case of a zero row sum matrix like $L[\bar{U}_\sigma^n]$ in the quantized scheme Eq. 2.48 on the preceding page the mass-exact matrix vector product (Eq. 2.28 on page 40) depends only on the differences of elements of $\bar{V}$ and the evaluation of $L[\bar{U}_\sigma^n]$

uses finite differences to approximate the gradient, such that the results are invariant to shifts in grey level as long as the addition of c does not produce a saturation in the number representation. □

- Reverse contrast invariance

$$S_n^{\mathcal{Q}}[-\bar{V}] \quad = \quad -S_n^{\mathcal{Q}}[\bar{V}]\,.$$

PROOF Similar to the argument above, the differences in Eq. 2.28 on page 40 allow a factorization of -1 and $L[\bar{U}_\sigma^n]$ is invariant to sign changes (Eqs. 2.41 and 2.44 on page 46). □

- Average grey level invariance

$$\mathcal{M}\big[S_n^{\mathcal{Q}}[\bar{V}]\big] \quad = \quad \mathcal{M}[\bar{V}]\,,$$

where $\mathcal{M} : \mathcal{Q}^{\#\Omega_h} \to \mathcal{Q}^{\#\Omega_h}$ is the averaging operator defined by

$$\mathcal{M}[\bar{V}] \quad := \quad \left(\frac{1}{\#\Omega_h}\sum_{\alpha\in\Omega_h}\bar{V}_\alpha\right)\bar{1}.$$

PROOF The preservation of this property was the reason for the introduction of the mass-exact matrix vector product which preserves the overall sum of elements for symmetric matrices with unit row sum (Eq. 2.30 on page 41). □

- Translation invariance

$$(S_n^{\mathcal{Q}}\circ\tau_p)[\bar{V}] \quad = \quad (\tau_p\circ S_n^{\mathcal{Q}})[\bar{V}]\,,$$

for translations $(\tau_p[\bar{V}])_\alpha := \bar{V}_{\alpha+p}$ with $p\in h\mathbb{Z}^2$ and supp $\tau_p[\bar{V}]$, supp $(\tau_p\circ S_n^{\mathcal{Q}})[\bar{V}]\subseteq\Omega_h$.

PROOF We obtain this invariance, because the process does not depend explicitely on the position of the nodes. But obviously, identical results are only obtained if we translate with multiples of the element grid size h. □

- Isometry invariance

$$(S_n^{\mathcal{Q}}\circ R)[\bar{V}] \quad = \quad (R\circ S_n^{\mathcal{Q}})[\bar{V}]\,,$$

for orthogonal rotations $R\in\mathbb{R}^{2\times 2}$ by multiples of $\pi/2$ defined by $(R[\bar{V}])_\alpha := \bar{V}_{R\alpha}$ with supp $R[\bar{V}]$, supp $(R\circ S_n^{\mathcal{Q}})[\bar{V}]\subseteq\Omega_h$.

PROOF As above, invariance is only valid if the nodes in Ω_h map one-to-one under R. □

- Extremum principle

$$\min_\beta \bar{V}_\beta \leq S^{\mathcal{Q}}_n[\bar{V}]_\alpha \leq \max_\beta \bar{V}_\beta\,. \tag{2.53}$$

PROOF It suffices to prove the inequalities for $n = 1$, because the semi-group property allows us to set $\bar{W} := S^{\mathcal{Q}}_{n-1}[\bar{V}]$ and then $S^{\mathcal{Q}}_n[\bar{V}] = S^{\mathcal{Q}}_1[\bar{W}]$, such that the result for general n follows by induction.

From the backward error analysis of the mass-exact matrix vector product we know that the quantized product corresponds to a perturbation of the original matrix (Eq. 2.33 on page 42). For our scheme (Eq. 2.48 on page 48) this means

$$S^{\mathcal{Q}}_1[\bar{V}] \;=\; A^-[\bar{V}] \boxdot \bar{V} \;=\; B^{A^-,\bar{V}} \cdot \bar{V} \tag{2.54}$$

with the symmetric unit row sum matrix

$$\begin{aligned} (B^{A^-,\bar{V}})_{\alpha\beta} &:= \lambda^{\alpha\beta}_\mu A^-_{\alpha\beta}, \qquad \beta \neq \alpha, \\ (B^{A^-,\bar{V}})_{\alpha\alpha} &:= 1 - \sum_{\beta:\beta\neq\alpha} \lambda^{\alpha\beta}_\mu A^-_{\alpha\beta}\,. \end{aligned} \tag{2.55}$$

If all elements of $B^{A^-,\bar{V}}$ are non-negative the inequalities follow

$$\min_\beta \bar{V}_\beta \cdot \underbrace{\sum_\beta B^{A^-,\bar{V}}_{\alpha\beta}}_{=1} \leq S^{\mathcal{Q}}_1[\bar{V}]_\alpha = \sum_\beta B^{A^-,\bar{V}}_{\alpha\beta} \bar{V}_\beta \leq \max_\beta \bar{V}_\beta \cdot \underbrace{\sum_\beta B^{A^-,\bar{V}}_{\alpha\beta}}_{=1}\,,$$

because $B^{A^-,\bar{V}}$ has unit row sum.

Now, we have to specify the condition under which all elements of $B^{A^-,\bar{V}}$ are non-negative. From the properties of the weighted stiffness matrix $L[\bar{U}^n_\sigma]$ on page 47 we know that all elements of A^- are non-negative under the assumption $\frac{\tau^n}{h^2} < \frac{1}{\max_\alpha |L_{\alpha\alpha}|} \leq \frac{3}{8}$. The roundoff error factors $\lambda^{\alpha\beta}_\mu$ (Eq. 2.23 on page 32) are always non-negative, such that the off-diagonal elements of $B^{A^-,\bar{V}}$ are definitely non-negative. For the diagonal elements we have the condition:

$$\begin{aligned} (B^{A^-,\bar{V}})_{\alpha\alpha} \;&\geq\; 1 - \max_{\beta:\beta\neq\alpha} \lambda^{\alpha\beta}_\mu \cdot \sum_{\beta:\beta\neq\alpha} A^-_{\alpha\beta} \;=\; 1 - \max_{\beta:\beta\neq\alpha} \lambda^{\alpha\beta}_\mu \cdot \frac{\tau^n}{h^2} L_{\alpha\alpha} \\ &\geq\; 1 - \frac{8}{3}\frac{\tau^n}{h^2} \max_{\beta:\beta\neq\alpha} \lambda^{\alpha\beta}_\mu \;>\; 0\,, \end{aligned}$$

where we use the bound on the diagonal elements $L_{\alpha\alpha}$ from Eq. 2.47 on page 47. For some rounding modes μ the correction factors $\lambda^{\alpha\beta}_\mu$ are bounded uniformly (Eq. 2.24 on page 32). For the modes rounding to zero $\mu = 0$ and nearest $\mu = \mathfrak{n}$ we thus obtain the time-step condition

$$\frac{\tau^n}{h^2} \;<\; \begin{cases} \frac{3}{8} & \text{if } \mu = 0 \\ \frac{3}{16} & \text{if } \mu = \mathfrak{n} \end{cases}\,. \tag{2.56}$$

Under this condition the quantized scale-space satisfies the extremum principle. □

- Lyapunov functionals
For $\bar{V} \in \mathcal{Q}^{\#\Omega_h}$ and a convex $r \in C[\min_\alpha \bar{V}_\alpha, \max_\alpha \bar{V}_\alpha]$, the functional

$$\Phi[n,\bar{V}] \;:=\; \sum_\alpha r\left(S_n^{\mathcal{Q}}[\bar{V}]_\alpha\right) \tag{2.57}$$

is a *Lyapunov functional*:

$$\Phi[0,\mathcal{M}[\bar{V}]] \;\le\; \Phi[n,\bar{V}]\,, \tag{2.58}$$
$$\Phi[n+1,\bar{V}] \;\le\; \Phi[n,\bar{V}]\,. \tag{2.59}$$

If r is strictly convex on $[\min_\alpha \bar{V}_\alpha, \max_\alpha \bar{V}_\alpha]$ then $\Phi[n,\bar{V}]$ is a strict Lyapunov functional:

$$\Phi[0,\mathcal{M}[\bar{V}]] = \Phi[n,\bar{V}] \quad\Longleftrightarrow\quad S_n^{\mathcal{Q}}[\bar{V}] = \mathcal{M}[\bar{V}]\,, \tag{2.60}$$
$$\Phi[n+1,\bar{V}] = \Phi[n,\bar{V}] \quad\Longleftrightarrow\quad S_n^{\mathcal{Q}}[\bar{V}] = \mathcal{M}[\bar{V}] \text{ or } B^{A^-,S_n^{\mathcal{Q}}[\bar{V}]} = 1\!\!1\,. \tag{2.61}$$

PROOF In the following we use the abbreviation $\bar{V}^n := S_n^{\mathcal{Q}}[\bar{V}]$. The first inequality 2.58 is a consequence of the convexity of r:

$$\begin{aligned}\Phi[0,\mathcal{M}[\bar{V}]] &= \Phi[0,\mathcal{M}[\bar{V}^n]] = \sum_\alpha r\left(\sum_\beta \frac{1}{\#\Omega_h}\bar{V}^n_\beta\right)\\ &\le \underbrace{\sum_\alpha \frac{1}{\#\Omega_h}}_{=1} \sum_\beta r\left(\bar{V}^n_\beta\right)\\ &= \Phi[n,\bar{V}]\,.\end{aligned} \tag{2.62}$$

The second inequality 2.59 will be proved under the time-step condition from Eq. 2.56 on the page before, as we require the equality $A^-[\bar{V}] \boxdot \bar{V} = B^{A^-,\bar{V}} \cdot \bar{V}$ (Eq. 2.54 on the preceding page). The condition guarantees that all elements of $B^{A^-,\bar{V}}$ (Eq. 2.55 on the page before) are non-negative, such that the row elements, which have row sum one, can be used as convex factors:

$$\begin{aligned}\Phi[n+1,\bar{V}] - \Phi[n,\bar{V}] &= \sum_\alpha r\left(\sum_\beta B^{A^-,\bar{V}}_{\alpha\beta}\bar{V}^n_\beta\right) - \sum_\alpha r\left(\bar{V}^n_\alpha\right)\\ &\le \sum_\beta \underbrace{\sum_\alpha B^{A^-,\bar{V}}_{\alpha\beta}}_{=1} r\left(\bar{V}^n_\beta\right) - \sum_\alpha r\left(\bar{V}^n_\alpha\right)\\ &= 0\,.\end{aligned} \tag{2.63}$$

The first consequence of the strict convexity of r (Eq. 2.60 on the facing page) follows immediately from the fact that equality in the convex estimate in the derivation 2.62 on the preceding page holds true if and only if all addends in the sum are equal, i.e. $\bar{V}^n = \mathcal{M}[\bar{V}]$.

Similar the second equivalence (Eq. 2.61 on the facing page) follows from the fact that equality in the convex estimate in the derivation 2.63 on the preceding page holds true if and only if all addends in the sum are equal, i.e. $\bar{V}^n = \mathcal{M}[\bar{V}]$, or there are no multiple addends, i.e. $B^{A^-,\bar{V}} = \mathbb{1}$. □

The alternative condition $B^{A^-,\bar{V}} = \mathbb{1}$ is the theoretic price we pay for the quantization. The matrix A^- itself has always several non-zero entries in a row, but depending on the current solution $\bar{V}^n$, all the multiplications with off-diagonal elements may round to zero, such that effectively we apply a unit matrix.

- Convergence

$$\exists N^{\bar{V}} \in \mathbb{N}_0, \bar{C}^{\bar{V}} \in \mathcal{Q}^{\#\Omega_h} : \forall m \geq N^{\bar{V}} : \qquad S_m^{\mathcal{Q}}(\bar{V}) = \bar{C}^{\bar{V}} \tag{2.64}$$

$$\lim_{\varepsilon^{\mathcal{Q}} \to 0} \|\bar{C}^{\bar{V}} - \mathcal{M}[\bar{V}]\| = 0\,. \tag{2.65}$$

Proof The function $r(x) := x^2$ is strictly convex. The sequence $\left(\Phi[n,\bar{V}]\right)_n$ is then positive, monotone decreasing (Eq. 2.59 on the facing page) and can take only finitely many values ($\mathcal{Q}$ is finite), such that

$$\exists N^{\bar{V}} \in \mathbb{N}_0 : \forall m \geq N^{\bar{V}} : \qquad \Phi[m,\bar{V}] = \Phi[N^{\bar{V}},\bar{V}]\,.$$

If we set

$$\bar{C}^{\bar{V}} := S_{N^{\bar{V}}}^{\mathcal{Q}}[\bar{V}]$$

then Eq. 2.61 on the preceding page implies $\bar{C}^{\bar{V}} = \mathcal{M}[\bar{V}]$ or $B^{A^-,\bar{C}^{\bar{V}}} = \mathbb{1}$. In both cases the application of $S_1^{\mathcal{Q}}$ will not change $\bar{C}^{\bar{V}}$ any more, such that Eq. 2.64 follows.

To specify how much $\bar{C}^{\bar{V}}$ differs from $\mathcal{M}[\bar{V}]$ we must closer examine the condition $B^{A^-,\bar{C}^{\bar{V}}} = \mathbb{1}$. For the condition to hold we must have (Eq. 2.55 on page 51):

$$\lambda_\mu^{\alpha\beta} = \lambda_\mu\left(A^-_{\alpha\beta}(\bar{C}^{\bar{V}}_\alpha - \bar{C}^{\bar{V}}_\beta)\right) = \frac{\mathrm{round}_\mu^{\mathcal{Q}}\left(A^-_{\alpha\beta}(\bar{C}^{\bar{V}}_\alpha - \bar{C}^{\bar{V}}_\beta)\right)}{\left(A^-_{\alpha\beta}(\bar{C}^{\bar{V}}_\alpha - \bar{C}^{\bar{V}}_\beta)\right)} = 0\,.$$

This means that $B^{A^-,\bar{C}^{\bar{V}}} = \mathbb{1}$ is equivalent to

$$\mathrm{round}_\mu^{\mathcal{Q}}\left(\left(\frac{\tau^n}{h^2}L[\bar{C}^{\bar{V}}_\sigma]\right)_{\alpha\beta}(\bar{C}^{\bar{V}}_\alpha - \bar{C}^{\bar{V}}_\beta)\right) = 0 \tag{2.66}$$

for all $\alpha \neq \beta$. This condition includes the case $\bar{C}^{\bar{V}} = \mathcal{M}[\bar{C}^{\bar{V}}]$, because then all differences $\bar{C}^{\bar{V}}_\alpha - \bar{C}^{\bar{V}}_\beta$ are zero. If g_2 is bounded from below by some positive constant, then the magnitude of two of the three weights of the diagonal ($|\alpha-\beta| = (1,1)$), the horizontal ($|\alpha-\beta| = (0,1)$) or the vertical ($|\alpha-\beta| = (1,0)$) factors in each element (cf. Eq. 2.45 on page 46) have a positive lower bound. The contributions of the element weights to $\frac{\tau^n}{h^2} L[\bar{C}^{\bar{V}}_\sigma]$ do not cancel out in the summation (Eq. 2.41 on page 45), as they are all required to be non-positive, resulting in some uniform lower bound c. Thus if all multiplications in Eq. 2.66 on the preceding page underflow to zero, it means that all $|\bar{C}^{\bar{V}}_\alpha - \bar{C}^{\bar{V}}_\beta|$ are smaller than $\frac{2}{c}\varepsilon^{\mathcal{Q}}$ within a grid element and smaller than $\frac{2}{c \cdot h}\varepsilon^{\mathcal{Q}}$ across the whole image. Consequently, the elements in $\bar{C}^{\bar{V}}$ converge to the same value as $\varepsilon^{\mathcal{Q}} \to 0$ and because the mass is preserved this value must be the average. □

The exact time-step $N^{\bar{V}}$ at which the process stops for a given $\varepsilon^{\mathcal{Q}}$, depends on the choice of the weight functions g_1, g_2. But in general we know that in areas with large gradients we will have $g_2 \gg g_1$ and in smooth areas $1 \approx g_2 \approx g_1$. In the first case the diffusion will continue, since $\bar{V}_\alpha - \bar{V}_\beta$ will be fairly large and one of the off-diagonal coefficients in $L[\bar{V}_\sigma]$ will continue the diffusion process. In the second case we will have almost a linear diffusion such that $L[\bar{V}_\sigma]_{\alpha\beta} \approx -\frac{1}{3}$ for all $\|\alpha-\beta\|_\infty = 1$. Thus if $\frac{\tau^n}{h^2} = \frac{3}{8(1+\delta_{\mu\in\mathfrak{n}})}$ (Eq. 2.56 on page 51) the diffusion stops roughly if all the differences $\bar{V}_\alpha - \bar{V}_\beta$ in a grid element become smaller than $8\varepsilon^{\mathcal{Q}}$, while the differences across the image may still cover the whole number range, since the width or height of an image is often larger than $1/(8\varepsilon^{\mathcal{Q}})$ (cf. Figure 4.11 on page 142). If all differences are small we can compensate this effect by simply scaling up all discrete differences, because such an operation corresponds to an implicitly enlarged time-step width in the explicit scheme. However, this is only possible if all differences are small, otherwise we would violate the *CFL* condition (Eq. 2.13 on page 25) and destabilize the process.

Therefore, image processing with low precision number systems is bad at transporting information across large distances on a fine grid. A multi-grid strategy must be used if this is required. But multi-grid solver are only successful if a certain precision $\varepsilon^{\mathcal{Q}}$ is available, otherwise the operations of restriction and prolongation produce artefacts due to roundoff errors. For the anisotropic diffusion the local smoothing properties are more important, because in practice the process runs only as long as the edges in the image are still well preserved. In the registration problem we will apply a multi-grid strategy, because correlation between distant regions will have to be examined.

The fact that the quantized scheme allows us to define Lyapunov functionals similar to the continuous setting guarantees that despite strong quantization we can maintain the qualitative smoothing properties of the continuous model. The following special Lyapunov functionals exemplify these properties:

- Vector norms $\|\bar{U}^n\|_p$ for $p \geq 1$.
  All such vector norms are decreasing with n and especially the energy $\|\bar{U}^n\|_2$ is reduced.
- Even central moments $\frac{1}{\#\Omega_h}\sum_\alpha(\bar{U}^n_\alpha - \mathcal{M}[\bar{U}]_\alpha)^{2p}$ for $p \in \mathbb{N}$.
  The decrease in central moments characterizes the smoothing of deviations from the

mean value. In particular the variance as the second central moment becomes consecutively smaller with n.

- Entropy $\quad -\sum_\alpha \bar{U}^n_\alpha \ln(\bar{U}^n_\alpha)$ for $\min_\alpha \bar{U}^0_\alpha > 0$.
  The entropy is constantly increasing with n and thus the quantized process subsequently reduces the information in the image.

2.4 Level-Set Segmentation

For the analysis and the implementation we will consider the front propagation under external forces $f(t,x) = c(t,x) + g_1(p(x)) + g_2(\|\nabla p(x)\|)$ exclusively (cf. Section 2.1.2.1 on page 19), such that the *level-set equation* obtains the form:

$$\begin{aligned} \partial_t \phi + f\|\nabla\phi\| &= 0 \;, \quad \text{in } \mathbb{R}^+ \times \Omega \,, \\ \phi(0) &= \phi_0 \,, \quad \text{on } \Omega \,. \end{aligned}$$

The choice of the functions g_1, g_2 depending on the image values $p(x)$ and its derivatives $\nabla p(x)$ is determined by the users input and should steer the evolution such that the desired segments are covered by the level-set function. The reason for the simple choice of the speed function, lies in the very low prerequisites on the functionality of the hardware used for the implementation of this problem.

2.4.1 Upwind Scheme

By introducing the Hamilton function $H(u) := f\|u\|$ we obtain a Hamilton-Jacobi equation

$$\partial_t \phi + H(\nabla\phi) = 0.$$

Using the explicit Euler time discretization (Eq. 2.12 on page 25) we have

$$\frac{\phi^{n+1} - \phi^n}{\tau^n} = H(\nabla\phi^n) \,,$$

where τ^n is the current time-step width. The upwind methodology [Engquist and Osher, 1980; Osher and Sethian, 1988] used in schemes for hyperbolic conservation laws (Eq. 2.7 on page 21) gives us a numerical flux function $g : \mathbb{R}^{2\times 2} \to \mathbb{R}$ which approximates H, i.e. $H(u(x)) \approx g(u(x-\varepsilon), u(x+\varepsilon))$. The flux g is constructed such that it uses values upwind the direction of information propagation (see Eq. 2.70 on the next page). For our equidistant grid (Figure 2.1 on page 27) we obtain the simple upwind level-set scheme:

$$\bar{\Phi}^{n+1}_\alpha = \bar{\Phi}^n_\alpha - \frac{\tau^n}{h} \bar{g}_\alpha(D^-\bar{\Phi}^n, D^+\bar{\Phi}^n) \tag{2.67}$$

$$D^+_\alpha \bar\Phi^n := \begin{pmatrix} \bar\Phi^n_{\alpha+(0,1)} - \bar\Phi^n_\alpha \\ \bar\Phi^n_{\alpha+(1,0)} - \bar\Phi^n_\alpha \end{pmatrix} \tag{2.68}$$

$$D^-_\alpha \bar\Phi^n := \begin{pmatrix} \bar\Phi^n_\alpha - \bar\Phi^n_{\alpha-(0,1)} \\ \bar\Phi^n_\alpha - \bar\Phi^n_{\alpha-(1,0)} \end{pmatrix} . \tag{2.69}$$

The discrete solution Φ^n is expected to approximate $\phi(\sum_{j=0}^{n-1} \tau^j)$. We implement the mirror boundary condition by copying for each n the border values $\bar\Phi^n_\alpha$ into a layer of ghost nodes surrounding the grid Ω_h. In this way we can always evaluate $D^\pm_\alpha \bar\Phi^n$ for all $\alpha \in \Omega_h$. It is important to note that the equidistant grid allows us to factorize $1/h$ from the gradient arguments to g, such that this large number can be compensated against τ^n.

For convex H a simple choice for the numerical flux g is the Enquist-Osher flux [Engquist and Osher, 1980]

$$\begin{aligned} \bar g_\alpha(\bar U, \bar V) &= \bar F^+_\alpha \sqrt{\|\bar U^+_\alpha\|^2 + \|\bar V^-_\alpha\|^2} + \bar F^-_\alpha \sqrt{\|\bar U^-_\alpha\|^2 + \|\bar V^+_\alpha\|^2} \\ X^+ &:= \max(X, 0), \qquad X^- := \min(X, 0). \end{aligned} \tag{2.70}$$

Using the explicit scheme we ensure stability by satisfying the *CFL* condition (Eq. 2.13 on page 25)

$$\frac{\tau^n}{h} \max_\alpha |\bar F_\alpha| < 1 . \tag{2.71}$$

Because we are using only external forces to steer the front propagation, we can precompute and normalize the velocities in advance, such that this condition directly restricts the size of the time-step width.

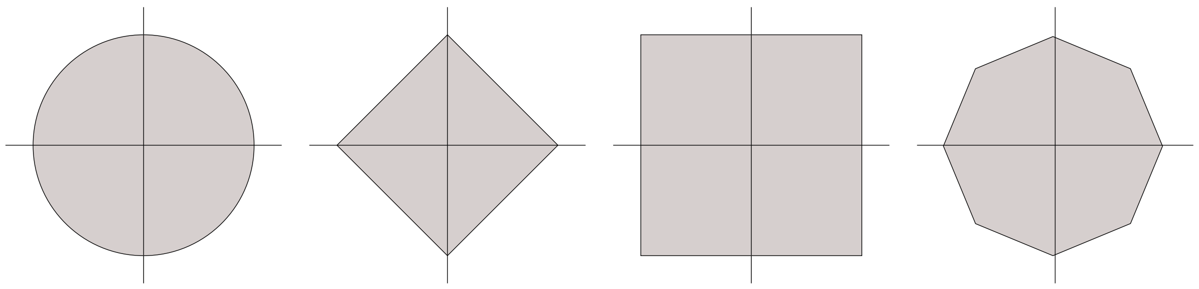
2.4.2 Quantized Propagation

The main difficulty in applying the upwind scheme (Eq. 2.67 on the preceding page) lies in the evaluation of the flux g (Eq. 2.70) under strong quantization. The involved Euclidean norm $\sqrt{\|U\|^2 + \|V\|^2}$ is a non-linear function of four values and thus very error-prone in fixed point calculations due to the aliasing effect (see examples on page 32 in Section 2.2.2.2). In particular, many evaluations would return zero, because the square function on $[0, 1]$ would underflow. In the presence of a two dimensional lookup mechanism one finds a remedy by computing the norm as a three step lookup, evaluating $\|U\|^2$ first, then $\|V\|^2$ and finally $\sqrt{\|U\|^2 + \|V\|^2}$. But if we rely on simple arithmetic operations then it is necessary to work with a piecewise linear approximation:

$$\|X\|_2 \approx \|X\|_{\text{lin}} := c\|X\|_1 + (1-c)\|X\|_\infty , \tag{2.72}$$

with $X \in \mathbb{R}^n$ and $c \in [0, 1]$. The norms $\|.\|_1$ and $\|.\|_\infty$ have the advantage that even with the quantized operators they can be evaluated exactly within the represented number range. Figure 2.2 on the facing page shows the unit spheres of the different norms.

Figure 2.2 The unit spheres of different vector norms in 2D. From left to right we see the Euclidean norm, 1-norm, infinity norm and the $\|.\|_{\text{lin}}$ norm, which is a convex combination of the previous two (Eq. 2.72 on the preceding page). It gives a good approximation and avoids the underflow in multiplications incurred in the evaluation of the Euclidean norm. The unit spheres are shown in 2D but the approximation takes place in 4D.



The convex combination guarantees that we have equality for the unit vectors $X = e_i$. For c we have various options. In an interpolative approach we choose $c = \frac{1}{\sqrt{n}+1}$, such that we obtain equality in Eq. 2.72 on the facing page for the vectors $(\pm\frac{1}{\sqrt{n}})_{i=0,\ldots,n-1}$. Another option is to approximate the volume of the n dimensional sphere by the unit balls of the norms $\|.\|_1$ and $\|.\|_\infty$ by choosing $c = (2^n - \frac{2\pi^{n/2}}{n\Gamma(n/2)})/(2^n - \frac{2^n}{\Gamma(n)})$. Finally, we can bound the relative error directly for uniformly distributed arguments in $V = [-1,1]^n$ by numerically minimizing $\int_V \left|\frac{\|X\|_2 - \|X\|_{\text{lin}}}{\|X\|_2}\right| \, dX$. Unfortunately, none of these choices can guarantee global properties of the approximation, because the arguments are not equally distributed in $[-1,1]^n$ and the distribution can vary significantly in different parts of the image. However, any of this approximations is good enough to be used in our low precision quantized computations, which themselves produce strong deviations from the continuous expansion, such that the effect of this approximation can only be seen in absolutely homogenous areas (Figure 4.3 on page 132).

With the new norm we obtain the quantized upwind scheme

$$\begin{aligned}
\bar{\Phi}^{n+1}_\alpha &= \bar{\Phi}^n_\alpha \ominus \bar{g}^{\text{lin}}_\alpha(D^-\bar{\Phi}^n, D^+\bar{\Phi}^n) \\
g^{\text{lin}}_\alpha(U,V) &:= (\tfrac{\tau^n}{h}\bar{F})^+_\alpha \odot \|(\bar{U}^+_\alpha, \bar{V}^-_\alpha)\|_{\text{lin}} \oplus (\tfrac{\tau^n}{h}\bar{F})^-_\alpha \odot \|(\bar{U}^-_\alpha, \bar{V}^+_\alpha)\|_{\text{lin}} \\
\|X\|_{\text{lin}} &:= c \odot \|X\|_1 \oplus (1-c) \odot \|X\|_\infty\,,
\end{aligned} \tag{2.73}$$

where we avoid an additional multiplication with $\frac{\tau^n}{h}$ by including the factor into the precomputed speed function and thus permanently satisfying the CFL condition (Eq. 2.71 on the facing page). We also realize that for the implementation we do not need any negative numbers, because the flux $\bar{g}^{\text{lin}}_\alpha$ requires only the absolute value of the differences $D^-_\alpha\bar{\Phi}^n, D^+_\alpha\bar{\Phi}^n$, such that it suffices to compute differences in $[0,1]$.

2.4.3 Asymptotic Behavior

As for the diffusion processes we cannot derive any bounds which relate the magnitude of the quantized solution $\bar{\Phi}$ directly to the continuous solution ϕ, because after, say 100 iterations, the roundoff errors in the multiplications in Eq. 2.73 on the page before for a typical machine epsilon of $\varepsilon^{\mathcal{Q}} = 3.9 \cdot 10^{-3}$ will easily corrupt any such result. But similar to the stopping criterion (Eq. 2.61 on page 52), we can find a condition describing the steady state of the evolution.

Obviously the evolution stops if $\bar{g}^{\text{lin}}_\alpha(D^-\bar{\Phi}^n, D^+\bar{\Phi}^n) = 0$ for every node $\alpha \in \Omega_h$. This will happen when all the multiplications $(\frac{\tau^n}{h}\bar{F})^\pm_\alpha \odot \|(\bar{U}^\pm_\alpha, \bar{V}^\mp_\alpha)\|_{\text{lin}}$ evaluate to zero. This is intended when $(\frac{\tau^n}{h}\bar{F})_\alpha$ is close to zero, since we have modeled the equation to behave in this way. However, underflows due to small differences $\bar{U}_\alpha = D^-_\alpha\bar{\Phi}^n, \bar{V}_\alpha = D^+_\alpha\bar{\Phi}^n$ are undesired. The approximate norm (Eq. 2.72 on page 56) already helps to minimize this effect by processing small arguments more accurately than a direct implementation of the Euclidean norm. But where the precomputed velocities $(\frac{\tau^n}{h}\bar{F})^\pm_\alpha$ are small the multiplications will underflow even for moderately small differences.

We can compensate this effect by rescaling $\bar{\Phi}$ such that the front steepens and the differences become larger:

$$\begin{aligned} \bar{\Phi}' &:= s_1 \cdot \bar{\Phi}, \quad s_1 \in \mathbb{N} \\ D^\pm\bar{\Phi}'^n &= s_1 \cdot D^\pm\bar{\Phi}^n \,. \end{aligned}$$

Applying such rescaling will saturate large positive and negative values, but this will have little effect on the front propagation, as it will only happen in regions far away from the front. Assuming exact computations the rescaling is equivalent to a scaling of the time-step width, since g^{lin} is positive 1-homogenous:

$$\bar{g}^{\text{lin}}(D^-\bar{\Phi}'^n, D^+\bar{\Phi}'^n) = \bar{g}^{\text{lin}}(s_1 \cdot D^-\bar{\Phi}^n, s_1 \cdot D^+\bar{\Phi}^n) = s_1 \cdot \bar{g}^{\text{lin}}(D^-\bar{\Phi}^n, D^+\bar{\Phi}^n)\,.$$

Therefore, the scaling with s is a natural parameter adaption when the evolution of the front threatens to stop. For the quantized scheme the multiplications in the evaluation of $\|.\|_{\text{lin}}$ are more accurate when the argument is scaled than when the result is scaled, because more digits of c will have an impact on the result. This is desirable since we do not want to scale up existing errors.

Additionally, we may also increase the time-step width directly by scaling up the factors $(\frac{\tau^n}{h}\bar{F})^\pm_\alpha$:

$$s_2 \cdot \left(\frac{\tau^n}{h}\bar{F}\right)^\pm_\alpha \odot \|(\bar{U}^\pm_\alpha, \bar{V}^\mp_\alpha)\|_{\text{lin}} = \left(\frac{s_2 \cdot \tau^n}{h}\bar{F}\right)^\pm_\alpha \odot \|(\bar{U}^\pm_\alpha, \bar{V}^\mp_\alpha)\|_{\text{lin}}, \quad s_2 \in \mathbb{N}$$

This may become necessary if all the differences $D^\pm_\alpha\bar{\Phi}^n$ along the front are fairly large, but the velocities are very small.

Thus by choosing s_1, s_2 sufficiently large we can always prevent an ubiquitous underflow in the multiplications $(\frac{\tau^n}{h}\bar{F})^\pm_\alpha \odot \|(\bar{U}^\pm_\alpha, \bar{V}^\mp_\alpha)\|_{\text{lin}}$. In this way we can ensure that similar to the continuous propagation the quantized front will finally stop only where the velocity equals zero. However, we cannot exclude the case that the propagation will stop for some time in some parts of the image, while it will continue in others, simply because the limited number range can handle only a restricted number of different velocities, e.g. if the segmented region expands with a high velocity to the left and it should simultaneously expand very slowly to the right, then the movement to the right can stop altogether if the difference between the velocities is too large, and it will resume as soon as the expansion velocity to the left has sufficiently decreased. So the described technique makes the best use of the limited number range by providing highest accuracy for the quickly moving parts of the interface and paying less attention to the slower parts, such as to minimize the overall difference to the continuous evolution.

2.5 Gradient Flow Registration

Our task in the registration problem is to minimize the energy between a template image T and a reference image R (see Section 2.1.3 on page 22)

$$\begin{aligned} E_\epsilon[u] &= \frac{1}{2}\int_\Omega |T_\epsilon \circ (\mathbb{1}+u) - R_\epsilon|^2 \,, \\ E'_\epsilon[u] &= (T_\epsilon \circ (\mathbb{1}+u) - R_\epsilon)\nabla T_\epsilon \circ (\mathbb{1}+u)\,, \end{aligned}$$

by a regularized gradient flow

$$\begin{aligned} \partial_t u_{\epsilon_k} &= -A(\sigma)^{-1}E'_{\epsilon_k}[u_{\epsilon_k}]\,, && \text{in } \mathbb{R}^+ \times \Omega \\ u_{\epsilon_k}(0) &= u_{\epsilon_{k+1}}(t_{\epsilon_{k+1}}) \,, && \text{on } \Omega \\ u_{\epsilon_K}(0) &= 0 \,, && \text{on } \Omega \end{aligned} \tag{2.74}$$

$$A(\sigma) := \mathbb{1} - \tfrac{\sigma^2}{2}\Delta\,, \tag{2.75}$$

which starts on a coarse scale ϵ_K and consecutively passes the results of the coarser scale ϵ_{k+1} as the initialization to the next finer scale ϵ_k until the final result $u_{\epsilon_0}(t_{\epsilon_0})$ on the finest scale ϵ_0 is obtained. The solution then describes the deformation under which the template correlates with the reference image. The parameter $\sigma \in \mathbb{R}^+$ controls the regularization of the gradient flow during the process.

2.5.1 Multi-Grid Discretization

To solve the gradient flow problem on a given scale ϵ we use the explicit Euler time discretization (Eq. 2.12 on page 25) with the operator $\mathcal{F}[\mathcal{C}^p_\sigma[u], v] = \mathcal{C}^p_\sigma[u] = A(\sigma)^{-1}E'_\epsilon[u]$:

$$\frac{u^{n+1}_\epsilon - u^n_\epsilon}{\tau^n_\epsilon} = -A(\sigma)^{-1}E'_\epsilon[u^n]\,,$$

where τ_ϵ^n is determined by Armijo's rule [Kosmol, 1991]:

$$\frac{E_\epsilon[u_\epsilon^n] - E_\epsilon[u_\epsilon^{n+1}]}{\tau_\epsilon^n \left\langle E'_\epsilon[u_\epsilon^n], A(\sigma)^{-1} E'_\epsilon[u_\epsilon^n]\right\rangle} \geq c\,, \tag{2.76}$$

for all $c \in (0, \frac{1}{2})$, where $\langle ., . \rangle$ denotes the L^2 scalar product. This allows an adaptive acceleration of the gradient descent towards a minimum.

For the space discretization we use again an equidistant grid Ω_h (Figure 2.1 on page 27) with linear Finite Elements (FEs), but this time it serves as the bottom of a whole hierarchy of equidistant grids $(\Omega_{h_l})_{l=0,\dots,L}$, where Ω_{h_0} is the finest and Ω_{h_L} the coarsest grid. The element length in the grids doubles in each step $h_l = 2^{l-L}$. To transfer data within the pyramid we use restriction $\rho_{l+1}^l : \Omega_{h_l} \to \Omega_{h_{l+1}}$ and prolongation $\pi_l^{l+1} : \Omega_{h_{l+1}} \to \Omega_{h_l}$ operators:

$$\begin{aligned} \rho_{l+1}^l(\bar{U})_\alpha &:= \sum_{\beta\in\Omega_{h_l}} a_{\alpha\beta}^l \bar{U}_\beta, \qquad \alpha \in \Omega_{h_{l+1}}\,, \\ \pi_l^{l+1}(\bar{U})_\beta &:= \sum_{\alpha\in\Omega_{h_{l+1}}} a_{\alpha\beta}^l \bar{U}_\alpha, \qquad \beta \in \Omega_{h_l}\,, \end{aligned} \tag{2.77}$$

where the factors $a_{\alpha\beta}^l$ are defined through the representation of a coarse Θ_α^{l+1} by fine Θ_β^l basis functions

$$\Theta_\alpha^{l+1} = \sum_{\beta\in\Omega_{h_l}} a_{\alpha\beta}^l \Theta_\beta^l, \qquad \alpha \in \Omega_{h_{l+1}}\,.$$

The grid hierarchy is used for an efficient representation of the different image scales $\bar{R}_\epsilon, \bar{T}_\epsilon, \bar{U}_\epsilon$. Images on the scale ϵ are represented on the coarsest grid Ω_{h_l} for which

$$\epsilon_k \geq c_h \cdot h_l \tag{2.78}$$

for some $c_h \in [\frac{1}{2}, 2]$ still holds true (cf. Figure 4.12 on page 144), i.e. we have a monotone mapping $l : \epsilon \to l(\epsilon)$. Thus computations on coarse scales are performed on small grids and only the last few scales represented on the finest grid Ω_{h_0} consume the full processing power. For very coarse scales the function l is bounded from above

$$\forall\epsilon : l(\epsilon) \leq L_0 < L\,, \tag{2.79}$$

to avoid the computation of initial deformations on very small grids, e.g. Ω_{h_L} is only a 2x2 grid. Typical choices for the initial grid are $L - L_0 \in \{2, 3\}$. Naturally, the multi-grid V-cycle (Eq. 2.82 on the facing page) is not affected by this bound and uses all grids in the hierarchy.

On scale ϵ we have to compute the update formula

$$\begin{aligned} \bar{U}_\epsilon^{n+1} &= \bar{U}_\epsilon^n - \frac{\tau_\epsilon^n}{h_{l(\epsilon)}} A_{h_{l(\epsilon)}}(\sigma)^{-1} \bar{E}'_\epsilon[\bar{U}_\epsilon^n]\,, \\ \bar{E}'_\epsilon[\bar{U}_\epsilon^n] &= (\bar{T}_\epsilon \circ (\mathbb{1} + \bar{U}_\epsilon^n) - \bar{R}_\epsilon) \cdot \left(h_{l(\epsilon)} \nabla_{h_{l(\epsilon)}} \bar{T}_\epsilon \circ (\mathbb{1} + \bar{U}_\epsilon^n)\right)\,, \end{aligned} \tag{2.80}$$

where the matrix $A_{h_{l(\epsilon)}}(\sigma)$ is the discrete counterpart of the regularization operator $A(\sigma)$ (Eq. 2.75 on page 59) in the linear FE on $\Omega_{h_{l(\epsilon)}}$ and we have factorized $1/h_{l(\epsilon)}$ to make use of the whole number range in the regularization of the gradient with $A_{h_{l(\epsilon)}}(\sigma)^{-1}$. We iterate this formula N_ϵ times until the update is sufficiently small

$$\left\| \frac{\tau_\epsilon^n}{h_{l(\epsilon)}} A_{h_{l(\epsilon)}}(\sigma)^{-1} \bar{E}'_\epsilon[\bar{U}_\epsilon^{N_\epsilon}] \right\|_2^2 \;\le\; \delta \,. \tag{2.81}$$

Since multi-grid solvers are the most efficient tools in solving linear systems of equations, the gradient smoothing $A_{h_{l(\epsilon)}}(\sigma)^{-1}\bar{E}'_\epsilon[\bar{U}_\epsilon^n]$ is performed as a multi-grid V-cycle with Jacobi iterations (Eq. 2.39 on page 45) as smoother and the standard prolongation and restriction operators (Eq. 2.77 on the preceding page). Indeed to ensure an appropriate regularization it turns out to be sufficient to consider only a single multi-grid V-cycle

$$\mathrm{MGM}_{l(\epsilon)}(\sigma) \;\approx\; A_{h_{l(\epsilon)}}(\sigma)^{-1} \,. \tag{2.82}$$

The cycle starts on the grid $\Omega_{h_{l(\epsilon)}}$ and runs up to the coarsest grid Ω_{h_L} and back. Thereby, only very few Jacobi smoothing iterations are applied on each grid.

2.5.2 Quantized Registration

From the discrete update formula (Eq. 2.80 on the facing page) together with the multi-grid smother (Eq. 2.82) we obtain the quantized scheme:

$$\begin{aligned}
\bar{U}_{\epsilon_k}^{n+1} &= \bar{U}_{\epsilon_k}^{n} \ominus \frac{\tau_{\epsilon_k}^n}{h_{l(\epsilon_k)}} \odot \mathrm{MGM}_{l(\epsilon_k)}(\sigma) \bar{E}'_{\epsilon_k}[\bar{U}_{\epsilon_k}^n], \qquad n = 0,\ldots,N_{\epsilon_k}-1\,,\\
\bar{E}'_{\epsilon_k}[\bar{U}_{\epsilon_k}^n] &= (\bar{T}_{\epsilon_k} \circledcirc (\mathbb{1} + \bar{U}_{\epsilon_k}^n) \ominus \bar{R}_{\epsilon_k}) \odot \left(h_{l(\epsilon)} \nabla_{h_{l(\epsilon_k)}} \bar{T}_{\epsilon_k} \circledcirc (\mathbb{1} + \bar{U}_{\epsilon_k}^n) \right)\,,\\
\bar{U}_{\epsilon_k}^0 &= \pi_{l(\epsilon_k)}^{l(\epsilon_{k+1})} \left(\bar{U}_{\epsilon_{k+1}}^{N_{\epsilon_{k+1}}} \right), \qquad k = K-1,\ldots,0\,,\\
\bar{U}_{\epsilon_K}^0 &= \bar{0}\,.
\end{aligned} \tag{2.83}$$

We have three nested loops in this scheme. The outer loop with index k runs from the coarse to the fine scale representations and uses the prolongation operator to transfer data onto finer grids, whereby the prolongation $\pi_{l(\epsilon_k)}^{l(\epsilon_{k+1})}$ is the identity if the scales ϵ_{k+1} and ϵ_k are represented on the same grid. While the deformation vector $\bar{U}_{\epsilon_k}$ is transported top to bottom, the image scales $\bar{R}_{\epsilon_k}, \bar{T}_{\epsilon_k}$ are generated bottom to top by applying the discrete filter operator

$$S_{h_{l(\epsilon_k)}}(\epsilon_k) \;=\; \mathrm{MGM}_{l(\epsilon_k)}(\epsilon_k) \;\approx\; A_{h_{l(\epsilon_k)}}(\epsilon_k)^{-1} \tag{2.84}$$

to the original images R, T (Eq. 2.9 on page 23):

$$\begin{aligned}
T_{\epsilon_k} &:= S_{h_{l(\epsilon_k)}}(\epsilon_k) T,\\
R_{\epsilon_k} &:= S_{h_{l(\epsilon_k)}}(\epsilon_k) R.
\end{aligned}$$

The middle loop with index n performs the gradient descent on a given scale until the change in data becomes sufficiently small (Eq. 2.81 on the page before). There are two computationally intensive parts in this loop. The first computes the energy gradient $\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]$, where the operation of a quantized offset access to the images $\bar{T}_{\epsilon_k}$ and $\nabla_{h_{l(\epsilon_k)}}\bar{T}_{\epsilon_k}$ denoted by '$\circledcirc$' is performed through a bilinear interpolation of the value from neighboring nodes. In the second part the multi-grid cycle regularizes the gradient, where the formulas for the restriction, prolongation (Eq. 2.77 on page 60) and the Jacobi smoother (Eq. 2.39 on page 45) are applied on each grid. Since the regularization operator $A(\sigma) = \mathbb{1} - \frac{\sigma^2}{2}\Delta$ corresponds to the *linear diffusion*, the matrix A used in the Jacobi iterations has constant entries. Then the convex reformulation (Eq. 2.49 on page 48) used for the non-linear diffusion is exceedingly efficient, as the constant values of the matrix $L_D^{-1}(L_D - L)$ are known in advance.

The inner loop has no explicit index, but determines for each update the maximal time-step width $\tau^n_{\epsilon_k}$ which satisfies Armijo's rule (Eq. 2.76 on page 60), i.e. we maximize τ in

$$\begin{aligned} E_{\epsilon_k}[\bar{U}^n_{\epsilon_k}] - E_{\epsilon_k}\left[\bar{U}^n_{\epsilon_k} - \tfrac{\tau}{h_{l(\epsilon_k)}}\mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]\right] \\ \geq\ c\tfrac{\tau}{h^2_{l(\epsilon_k)}}\left\langle \bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}], \mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]\right\rangle_{h_{l(\epsilon_k)}} . \end{aligned} \tag{2.85}$$

In the above formula only the energy $E_{\epsilon_k}\left[\bar{U}^n_{\epsilon_k} - \frac{\tau}{h_{l(\epsilon_k)}}\mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]\right]$ which depends non-linearly on τ needs to be recomputed iteratively. The computation of the energy $E_{\epsilon_k}[.]$ and the lumped scalar product $\langle .,.\rangle_{h_{l(\epsilon_k)}}$ requires the summation of values across the whole image, which may be difficult to realize in some hardware architectures. Applying a local addition operator similar to the restriction operator ρ^l_{l+1} until a 1-node domain remains, which can be quickly retrieved, offers a resort to this problem.

2.5.3 Quantized Descent

Since the non-convex energy $E[u]$ has many local minima in the energy landscape, we need a two-fold regularization procedure, where the gradient regularization with $A(\sigma)^{-1}$ guarantees the existence of a unique solution of the gradient flow on a given scale (Eq. 2.8 on page 23) and the descent through the scales imitates an annealing process. Because of this two-fold regularization we are not interested in the exact solution on a given scale and use a fast approximation with a multi-grid V-cycle (Eq. 2.82 on the page before) of the regularization operator $A(\sigma)^{-1}$ and stop the iterations of the gradient flow (Eq. 2.80 on page 60) fairly quickly as soon as the update changes fall under a certain bound (Eq. 2.81 on the page before). This crude approximations are legitimate, as only the regularization effect and not the exact solution of the process is required. This means that the quantization of the procedures also does not change any qualitative behavior, since, in the view of a backward error analysis, it corresponds to a slightly different choice of the regularization operator, smoothing operator for the images and stopping condition for the update formula. The effect of the quantization may become problematic, only if the quantized version of the linear diffusion, which is used for both the regularization and the smoothing operator, does not perform the qualitative task of smoothing any more.

The analysis of the properties of the quantized diffusion processes in Section 2.3.3 on page 49 reveals that by using a mass preserving scheme (Eq. 2.48 on page 48) we obtain Lyapunov sequences of diffused images. This means that the process retains such qualitative properties as extremum principle, decreasing energy, decreasing even central moments or increasing entropy despite the quantized computing. The major shortcoming is the disability of the quantized process to equalize intensity differences across large distances on a single scale. But this is not required since the use of the multi-grid V-cycle for the regularization and the multi-scale strategy permit the exchange of information across large distances by transporting it through the grid hierarchy.

2.6 Data-Flow

For the sake of a common discretization approach, Section 2.2.1 on page 24 has already outlined the general formal structure of the discussed image processing PDEs (Eq. 2.10 on page 24):

$$\begin{aligned} \partial_t u + \mathcal{F}[\mathcal{C}^p_\sigma[u], u] &= 0 \,, & \text{in } \mathbb{R}^+ \times \Omega \,, \\ u(0) &= u_0 \,, & \text{on } \Omega \,, \end{aligned}$$

where the non-linear *classifier* $\mathcal{C}^p_\sigma[u]$ locally classifies u with respect to the desired evolution and the linear operator $\mathcal{F}[u, v]$ performs the composition of the classification result with the unknown (Eq. 2.11 on page 25).

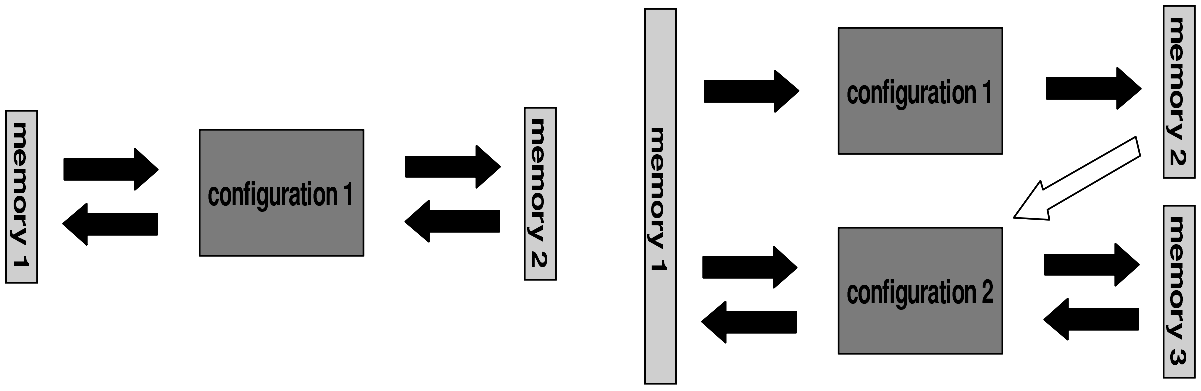
In the following we want to analyze which data-flow is needed to implement the high level structure of the iterative algorithms and the low level computations which evaluate the operators $\mathcal{C}^p_\sigma[u]$ and $\mathcal{F}[u, v]$. The importance of a regular data-flow stems from the characteristics of memory chips (Section 3.1 on page 71).

2.6.1 Explicit and Implicit Schemes

Similar time (Section 2.2.1.1 on page 25) and space (Section 2.2.1.2 on page 26) discretization schemes applied to the common PDE model result in similar discrete iterative schemes:

problem	equation	scheme	type
denoising	Eq. 2.37 on page 44	$\bar{U}^{n+1} = A^-[\bar{U}^n_\sigma] \cdot \bar{U}^n$	explicit
denoising	Eq. 2.38 on page 45	$A^+[\bar{U}^n_\sigma] \cdot \bar{U}^{n+1} = \bar{U}^n$	implicit
segmentation	Eq. 2.67 on page 55	$\bar{\Phi}^{n+1}_\alpha = \bar{\Phi}^n_\alpha - \frac{\tau^n}{h} g(D^-_\alpha \bar{\Phi}^n, D^+_\alpha \bar{\Phi}^n)$	explicit
registration	Eq. 2.80 on page 60	$\bar{U}^{n+1}_\epsilon = \bar{U}^n_\epsilon - \frac{\tau^n_\epsilon}{h_{l(\epsilon)}} A_{h_{l(\epsilon)}}(\sigma)^{-1} \bar{E}'_\epsilon[\bar{U}^n_\epsilon]$	implicit

Figure 2.3 High level data-flow of the PDE solvers for the explicit (left) and the implicit (right) discrete schemes.



The schemes have been labeled as being of explicit or implicit data-flow type. In case of the explicit iteration the new solution for the time point $n+1$ is directly computed from the current solution at time point n, while the implicit iteration requires to solve a linear equation system for each time-step to obtain the new solution. Notice that the scheme for the gradient flow registration uses an explicit time discretization, but the iterative scheme is of implicit type, because the regularization with $A_{h_{l(\epsilon)}}(\sigma)^{-1}$ corresponds to solving a linear equation system.

The explicit schemes are very attractive from the point of high level data-flow structure, because we need only one configuration to process the stream of data from $\bar{U}^n$ to $\bar{U}^{n+1}$ (Figure 2.3, left). To implement the whole time step loop we need only to alternate the source and destination memory addresses.

As we use iterative solvers for the solution of linear equation systems, practically, the implicit type requires an additional loop inside the time step loop. This means that the high level data-flow requires two different configurations of the PEs (Figure 2.3, right). Thinking of the usual software frameworks, the implementation of nested loops with different operations within the inner loop does not seem to impose any problems. But for the sake of efficiency we should rather think of the problem in terms of chip design, i.e. if we have two configurations, either the chip must be larger to hold both of them, or the functionality of the chip must allow some reconfiguration to perform one task or the other. In the first case the costs lie in the greater size of chip as compared to the explicit scheme, in the second the hardware must be more flexible and time is spent on the reconfiguration of the chip. We will elaborate on this thoughts in the next chapter.

We have seen that the type of the iterative schemes implies a different high level data-flow. But once established, it can be reused for all problems of the same type, since only the individual PEs which evaluate the operators $\mathcal{C}^p_\sigma$ and $\mathcal{F}$ must be replaced. This means for a hardware architecture that the general data-flow structure may be hard-coded and only the computational kernels require programmability. But even in fully reconfigurable architectures it has the advantage that only a few parts must be reconfigured for a new task, which saves both reconfiguration time and complexity of the overall design.

2.6.2 Data Locality

The low level data-flow in the computation of the operators $\mathcal{C}_\sigma^p$ and $\mathcal{F}$ also bears similarities between the problems as the access pattern to other nodes in the computation of a new node value is always restricted to a small neighborhood whether it occurs in the application of sparse band matrices, in the computation of spatial derivatives or as local offset accesses with bilinear interpolation of the values. In each case a certain number of the values from a small input neighborhood of a node is used to compute a single new value for the current node. This data locality can be efficiently exploited in the implementation of the algorithms if mesh nodes are stored closely together in memory and the access pattern to the input neighborhood is the same for each node. On equidistant meshes, that we use, this can be easily achieved, but on unstructured or adaptive meshes it is a difficult problem, and the effort to retain data locality in memory sometimes outweighs the gains from the fast access to node neighbors.

Although, in the general case, the classifier $\mathcal{C}_\sigma^p[u]$ may globally and non-linearly depend on the input data p or the current state u, this is almost never the case in practical applications, since it increases the computational load by orders of magnitude. Instead, far less time consuming evaluations of single global properties (e.g. norms) of functions on the whole domain are used to control the processes. The computation of such properties may be implemented in a way which does not violate the general assumption of data locality in computations (see the implementations on pages 146 and 178). But even with the knowledge of local data access in all computations, the question still arises which hardware architectures are appropriate for such processing and which quantitative parameters of the algorithms influence the choice of an optimal hardware platform.

2.6.3 Local Computations

Three important properties of the PDE algorithms are the size of the input neighborhood, the computational intensity and the write-read delay.

If the size of the ***input neighborhood*** is large, i.e. many neighboring values are used in the computation of a new node, then architectures with flexible *caches* and data routing are advantageous, because they can store the entire neighborhood of a node internally and deliver all values in parallel to the PEs. Ideally, each value is transferred only once to the cache, but read multiple times from the cache. If the size of the input neighborhood is small then less flexibility is needed and small automatic caches can reuse the data. In this case the raw *bandwidth* of the memory system and the *throughput* of the PEs are more important. Our algorithms usually work with a small neighborhood of 3 by 3.

If the ***computational intensity*** is high, i.e. complex, time consuming operations must be performed on each node, then architectures with many internal registers and long pipelines are advantageous, because the registers will hold intermediate results very close to the PEs, and the long pipeline will constitute a kind of super computational unit, which can perform many operations simultaneously. We speak of *parallelism in depth*. Ideally, the registers are integrated into the pipeline which is so long, that it contains all the PEs necessary for the computation of the new node value, such that a new value is output in every clock cycle. In case of low computational intensity the *bandwidth* of the memory system becomes the limiting factor and should be as high as possible. Most of the operations in the discrete solvers of our applications have a moderate to low computational intensity.

If the ***write-read delay*** of node values is high, i.e. newly computed values are not immediately needed in the subsequent computations, then architectures with many PEs perform best, because several node values can be processed in parallel. We speak of *parallelism in breadth*. Ideally, each pixel in an image is assigned its own PE, but this is too costly in almost all cases. For low write-read delays fast sequential processing with local caches for the reuse of newly computed values are more advantageous. As we have chosen discrete schemes and linear equation solvers which compute the components of the solution vector independently of each other for each time step, the write-read delay is high. Together with the regular data-flow, the parallelism in breadth is the key to the fast implementations of the solvers.

In practice, we usually cannot afford to set up a specific hardware which exactly meets the requirements of our application. Instead we use hardware platforms which are available at reasonable prices and serve our purpose best. Therefore, the next chapter will discuss the performance characteristics of various hardware platforms and the following presents the implementations on the chosen architectures.

2.7 Conclusions

We have examined PDE models for the image processing tasks of image denoising, segmentation and registration under the effects of roundoff errors in strongly quantized number systems. In general, even high precision floating point formats such as long double s63e15 do not immunize our computations against disastrous effects of roundoff errors (Section 2.2.2.1). For low precision fixed point computations the roundoff effects gain significant influence even more easily (Section 2.2.2.2), such that it becomes impossible to guarantee numerical stability for iterative schemes in neither deterministic nor probabilistic sense (Section 2.2.2.3). The price for the high performance of the implementations seems to be a loss of any quantitative bounds on the error resulting from the strong quantization.

However, control over the qualitative behavior of the low precision algorithms can be retained if special care is put into the factorization and aggregation of numbers on different scales (Sections 2.3.1, 2.4.1, 2.5.1), and the implementation of interactions between different node values (Section 2.2.3.3). The former requires an underlying equidistant grid, which is good for parallelization but inhibits adaptivity, although arrangements of equidistant sub-grids also provide adaptive concepts to some extent (Section 2.2.1.2). The latter, the careful interaction between node values, asks for special implementations of matrix vector products and prohibits some local accuracy optimization (Section 2.2.3.4) in favor of global mass conservation. The error analysis reveals that this scheme is well-suited for most image processing applications as it exhibits the highest accuracy in the vicinity of edges (Section 2.2.3.5).

Especially the diffusion processes benefit from the mass-exact matrix vector product as almost all of the properties of the continuous scale-space (Section 2.1.1.2) can be preserved in the discrete-quantized setting (Section 2.3.3). Thus, without the knowledge of any error bounds we can still be sure that the quantized algorithm behaves similar to the continuous model. In case of the segmentation we can ensure, that the front propagation will not stop too early due to multiplicative underflow, by adapting the time-step width appropriately (Section 2.4). Finally, the registration (Section 2.5) is fairly insensitive to quantization errors because the process aims at a hierarchically regularized descent in which the accuracy of each intermediate step is not so relevant. The qualitative properties of the decent, on the other hand, are guaranteed by the robust diffusion schemes used in the regularizations. Besides the pursuit of reliable results we have also payed attention to the resulting data-flow of our schemes. Data locality and different forms of parallelism (Section 2.6) can be exploited by various hardware architectures (Chapter 3) and are crucial for the efficiency of the later implementations (Chapter 4).

Recapitulating, we have seen that one can control the qualitative behavior of certain PDE processes without high local accuracy or global error bounds. Image processing applications lend themselves to this kind of fast low precision algorithms, because usually there is no a-priori ground truth for their results. The characteristics of the image evolution are more important than the individual values. By concentrating the available precision on the treatment of the characteristic image features in a given application one can therefore obtain satisfactory results at very high performance.

3 Data Processing

Contents

Figures

Tables

In this chapter we consider the general problem of data processing and recall some of the corresponding concepts in hardware design. In particular, we discuss the characteristics of data access such as latency and bandwidth, and the difference between instruction-stream-based (ISB) and data-stream-based (DSB) computing. We also examine the options of parallelization and how it can improve the different aspects of performance. Then, we consider the status quo of various hardware architectures and conclude with some thoughts on the future of computing architectures.

But first let us explain the general division between storing and computing silicon. The problem of data processing seems very simple: we have some data and want to apply a certain operation to it. Then, we either continue to apply another operation, output or visualize the result. We need some piece of hardware, some ***processing elements (PEs)***, which can perform the desired operation and some memory which holds the data. Obviously, it would be fastest to have the data already at the input of the PEs and activating them would deliver the result. In fact, such solutions exist for small data amounts. However, for large data blocks (e.g. an image with float values $1024^2 \cdot 4B = 4$MiB) the required memory size is very large. This means that we cannot afford to have a processing path for each data component, i.e. the same PE will have to work on different components consecutively, and due to the fabrication process the memory for the data cannot be placed on the same device as the PEs, i.e. it must reside in an external chip outside of the chip containing the PEs (Section 3.2.5 on page 96 discusses exceptions).

Being forced into an arrangement where the PEs reside on one chip and the main memory on an other we can separately analyze the performance characteristics of memory chips and the different paradigms of performing the actual computations.

3.1 Data Access

Moore's prediction from 1965 of an exponential growth of transistor count on the same size of silicon area, still holds true. This is usually referred to as ***Moore's Law***, although Moore was more interested in transistor cost than pure count [Moore, 1965]. The exponential growth did not take place at the same rate in the past either, with a 1 year transistor count doubling till 1974, a 2 year doubling since then and probably with a 3 years period in the following years [SEMATECH, 2003]. But from the numbers of transistors alone, it is not yet clear how the characteristics of the produced chips have changed.

For memory chips we have to consider three characteristics: ***memory size***, the amount of data that can be stored on the chip; ***bandwidth***, the amount of data that can be transported per second; ***latency***, the delay from the data request to the data availability. Moore's Law merits have been mainly directed into size enlargement, which has experienced exponential growth, but bandwidth and latency have improved only slowly. This is especially bad for the latency which improves very slowly and has already become the main degrading factor in PC performance. Special memory access modes and interleave strategies help to hide latency, but they require a regular data-flow, an important aspect of our solvers described in Section 2.6 on page 63. Although improvements around the memory core on the chip and the whole memory system have realized an exponential growth of the system bandwidth, this is slower than the growth in data bandwidth requirements of the PEs. So the gap between the available and the required bandwidth is widening, making bandwidth a performance bottleneck even when almost all latencies can be hidden. These memory problems are known as the ***memory gap*** and many ways of attacking it have been thought of, but those which are cost efficient for the mass production do not suffice to catch up with the development of the computing resources [Wilkes, 2000].

The next sections explain the current technology behind the memory chips and the reason for this important imbalance between the performance of the PEs and the memory system. Naturally, this problem has been anticipated for some time and many different technological improvements or even new memory concepts exist or are pending. Here, we can only sketch the technological mainstream. For more details and alternative technology we refer to [Alakarhu and Niittylahti, 2004; Davis et al., 2000a,b; Desikan et al., 2001; Keeth and Baker, 2000; Sharma, 2002a,b].

3.1.1 Random Access Memory

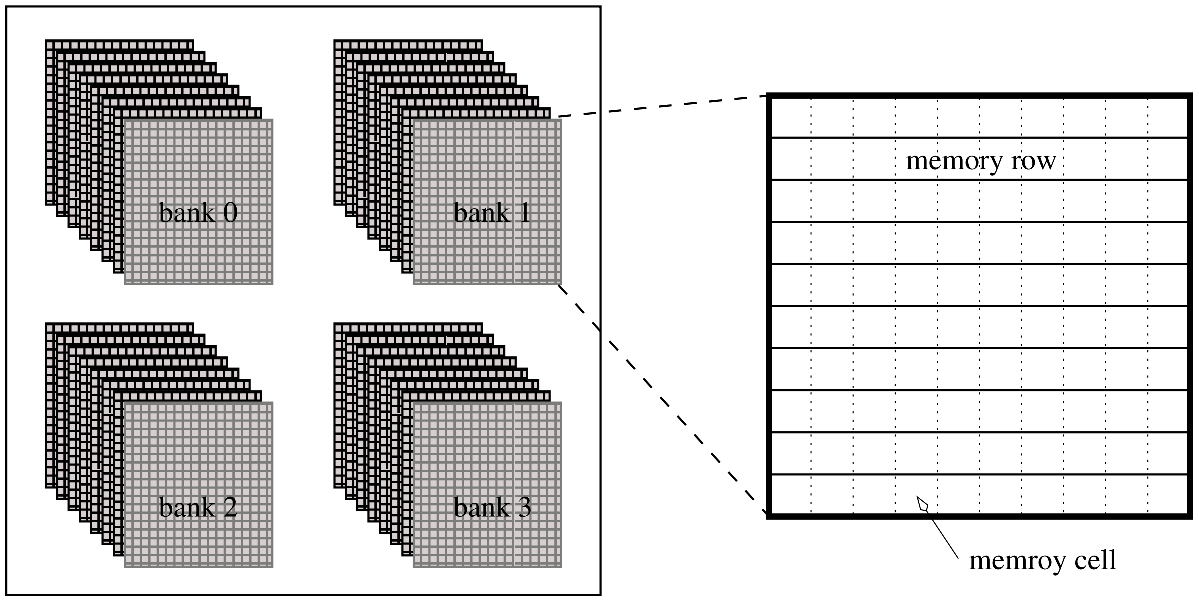
Random Access Memory (RAM) denotes a type of memory which allows access to the stored data in a random order, unlike e.g. First In First Out (FIFO) memory, which prescribes the order of the data access. As the access is random we must specify which part of the memory we want to access, i.e. we must specify a ***memory address***. Logically the address space is one dimensional, but since we use two dimensional silicon wafers for chip production, memory cells are arranged in a two dimensional ***memory array*** and the address is split into a column address and a row address. A ***memory cell*** in the memory array is a circuit which stores a bit, i.e. the binary 0 or 1. In Static RAM (SRAM) a memory cell consists of either 6 transistors (robust, power efficient) or only 4 (compact, area efficient) and the bit is stored permanently, until a write operation changes it or the power is switched off. A DRAM memory cell requires only 1 transistor and 1 capacitor, but the charge must be refreshed every few milliseconds, otherwise the information is lost. Despite the complication of the ***memory refresh***, DRAM is preferred by memory chip manufacturers, because it yields a much larger memory size for the same number of transistors. Therefore, in a typical system DRAM is used for the large main memory, while SRAM serves as embedded and low power memory (see Section 3.1.3 on page 81). As SRAM is more architecture specific the designs differ stronger than the typical DRAM design used for mass production.

Since from each address we usually read more than just one bit, several memory cells are grouped together to a ***memory depth*** of $2^0, \ldots, 2^6$ bit. The value 8 is nowadays typical for the main memory of a PC. More would be desirable, since the system accesses memory in larger chunks anyway (see Section 3.1.2 on page 79), but higher depth also means more data pins on the chip, which raises the costs. On the other hand, high density memory chips are cost efficient, because one needs fewer chips for the same amount of memory. So instead of having more than 8 bit memory depth, we have several ***memory banks*** on a chip, typically 4, and need only 2 pins to choose among the banks (Figure 3.1 on the facing page). The reason for having 4 banks rather than quadrupling the address space of the chip, which also requires only 2 more pins, is lower power consumption (only one bank is active at a time) and shorter strobes. Section 3.1.1.2 on page 76 describes another advantage related to the memory access. The standard size notation b-bank gMi $\times$ d for a memory chip, says that the chip has g mebi $(= 2^{20})$ memory cell groups with a depth of d bit arranged in b banks and thus holds $\frac{g \cdot d}{8}$ MiB or $\frac{g \cdot d}{8 \cdot b}$ MiB per bank, e.g. a 4-bank 32Mi $\times$ 8 chip holds 32MiB in 4 banks of 8MiB (see Section 1.2.4 on page 7 for the definition of Mi, mebi, etc.).

3.1.1.1 Latency

The pins of a memory chip consist of address, data, and few control and power supply pins. DRAM chips additionally save on address pins by receiving the column and row address consecutively and not simultaneously as for typical SRAM. A read access at a certain address is performed by opening a memory row (or rather *memory page*, see Section 3.1.2 on page 79 for the difference), i.e. measuring and amplifying the charge of the capacitors in the row with so called ***sense amps***. Then the data starting at the addressed column is selected from the sense amps. Obviously, once a row is open one can quickly obtain other data from the same row by only specifying a new column address and accessing it from the sense amps. This is important since each of the operations in the process involves a delay and if there are less operations to be performed there is less delay. If data from a different row is needed, the current row must be closed, i.e. the content of the sense amps is transferred back to the row in the memory array and a pre-charged must prepare the logic for the measuring of the charges in the new row. The new row can then be opened.

Figure 3.1 Internal structure of a memory chip. The left image shows a 4-bank 8 bit deep chip. To the right we see the memory array with the rows containing the memory cells.



There are two types of latencies in the above operations: ***access time***, the time period necessary to complete an operation, and ***cycle time***, the smallest possible time period between two states of the same signal. Table 3.1 on the next page lists important latencies and defines their symbols. Because of the different combinations of these latencies, previously, a processor had to switch into a wait state and expect the arrival of the requested data. Current Synchronous DRAM (SDRAM) chips run synchronous to a common clock cycle such that latencies are multiples thereof. Consequently, the overall latency depending on the access mode is known as a number of clock cycles in advance, so that the processor can perform some other task and knows when the requested data will be at its input. This again points out the efficiency of an algorithm with a regular data-flow, where the input address can be issued in advance while the processor is busy with previous data, so that the new data arrives on time to be processed. Figure 3.2 on page 75 shows the timings of different read accesses and the devastating impact of the latencies in case of random access reads. We distinguish three modes:

- row hit: we read data from an already open row,
  latency: tCL ,
- row miss from closed: we open a row and read data,
  latency: tRCD + tCL ,
- row miss from open: we close the current row, open a new row and read data,
  latency: tRP + tRCD + tCL .

The standard chip timing notation SDRAM tCLK tCL–tRCD–tRP–tRAS gives four of the latencies from Table 3.1 on the following page and the minimal clock cycle time (tCLK), or

Table 3.1 Simplified memory access latencies. In reality memory timings are more complicated, because for every signal and transition from inactive to active there is a setup time which allows the signal to stabilize and a hold time in which it can be accessed. See also Figure 3.2 on the facing page for a schematic visualization of the latencies.

latency	type	definition
row to column delay (tRCD)	access time	time from row active to column active
column access latency (tCL)	access time	time from column active to data output
column active time (tCAS)	cycle time	minimal active time of the column signal
column precharge (tCP)	cycle time	minimal inactive time of the column signal
row active time (tRAS)	cycle time	minimal active time of the row signal
row precharge (tRP)	cycle time	minimal inactive time of the row signal

sometimes the maximal frequency (1/tCLK) instead, e.g. SDRAM 10ns 2-2-2-6 or SDRAM 100MHz 2-2-2-6. Notice that the latencies are usually given as multiples of the tCLK, but in reality the latencies are sums of different internal delays, which round up to the next clock cycle for a given frequency. So if the above chip could also be run at 133MHz the resulting latencies might be very different e.g. 2-2-2-6, 3-2-2-7, 3-3-3-7, and we would need to consult the data sheet of the chip to find out which applies. The first three numbers cannot exceed 3, however, because from the first notation we know that the latency times for the chip are smaller than $2 \cdot 10\text{ns} = 20\text{ns}$ and $3 \cdot 7.5\text{ns} > 20\text{ns}$, where 7.5ns corresponds to the 133MHz frequency.

The delays tCAS and tCP do not show up in the timing notation, because in todays pipelined memory chips they are hidden by the ***burst mode***. Thereby, a certain number of columns called burst length (BL), typically 4 or 8, starting from the supplied column address is output one by one, without supplying a new address. A pipelined design can prepare the burst of the next BL columns while the data from the current columns is output. If

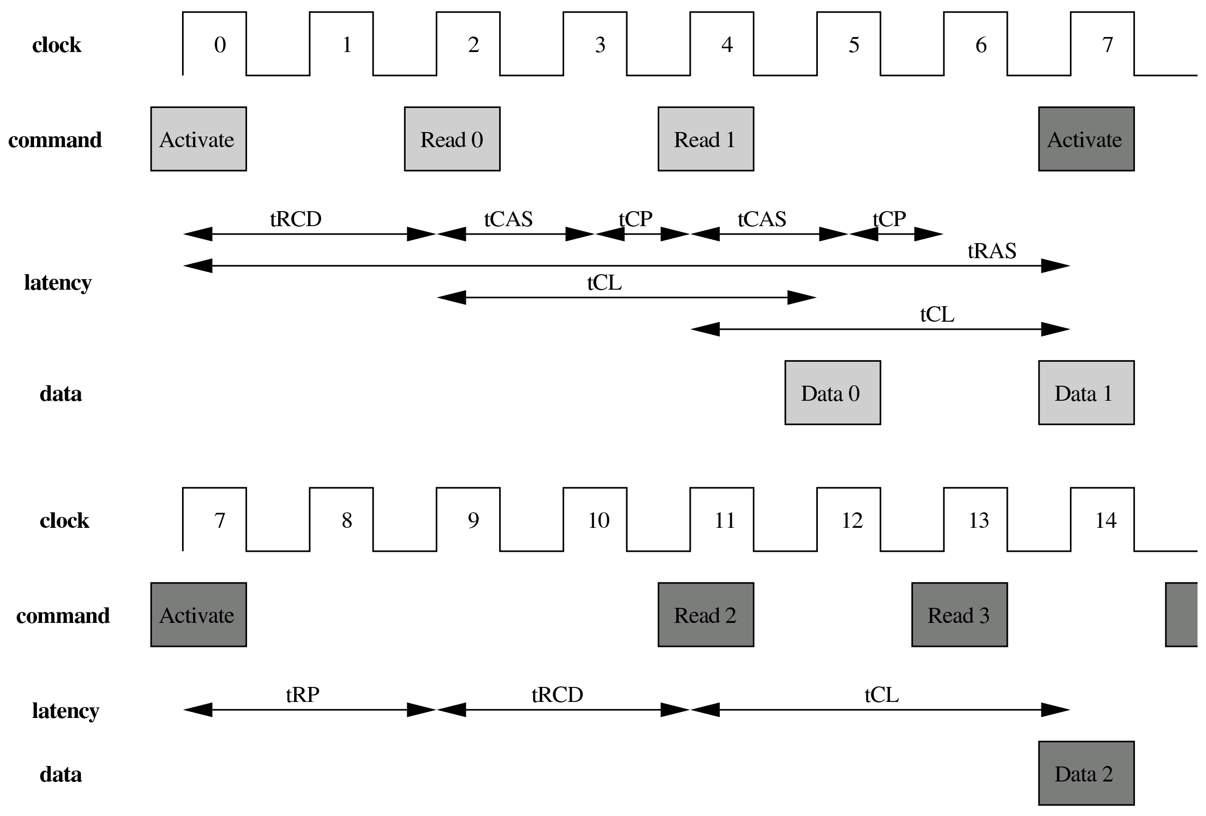
$$\text{tCAS} + \text{tCP} \quad \leq \quad \text{BL} \cdot \text{tCLK}, \tag{3.1}$$

then we can get a seamless stream of data from an open row after the initial tCL (Figure 3.3 on page 76). The other cases yield the following latencies:

mode	latency	1 packet	4 packets	8 packets
row hit	tCL	60ns (17%)	60ns (67%)	100ns (80%)
row miss from closed	tRCD+tCL	80ns (12.5%)	80ns (50%)	120ns (67%)
row miss from open	tRP+tRCD+tCL	100ns (10%)	100ns (40%)	140ns (57%)

The example times are computed for SDRAM 10ns 2-2-2-6 with BL=4 and 1,4 or 8 consecutive packets being read. The percentage numbers reflect the ratio of the ***sustained bandwidth***,

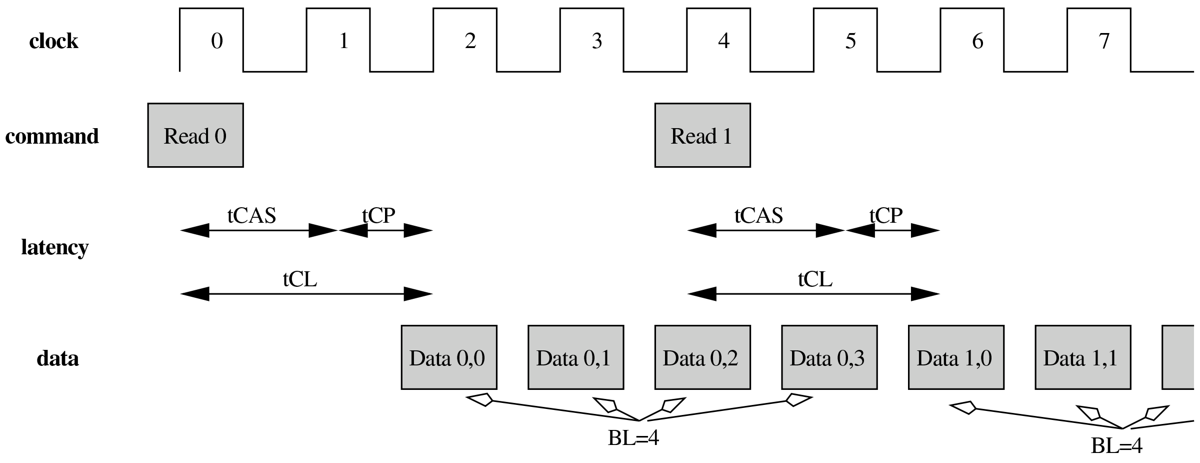
Figure 3.2 Timings in a SDRAM chip for different read accesses. Table 3.1 on the preceding page defines the latencies. The `Activate` command activates a bank and opens a row by specifying a row address on the address bus. The `Read` command reads data from the open row at the specified column address. Commands and data related to the same row have a common color.
In the beginning all rows are closed. The first `Activate` command opens a row and it remains open during the cycles 0-6. The second `Activate` closes the current row and opens a new one. Accordingly, the first `Read` is a row miss from closed, the second a row hit, the third a row miss from open and the fourth a row hit again.



the bandwidth achieved in the respective mode, and the ***peak bandwidth***, the maximal achievable bandwidth assuming no latencies. Obviously, this ratio improves the more consecutive packets from a given address are requested, whereby we assume that all packets lie in the same row. Therefore, algorithms exerting a linear memory access pattern perform much better than algorithms with random memory access. In the table above the performance factor between the worst and best case is as much as 8, and it would further grow to almost 10 if we requested more consecutive packets.

One could argue that if we want only one packet from each row then it would be more reasonable to set BL to 1. But since each row must stay open for tRAS (=6 in this case), the row miss from open would take tRP+tRAS time plus one tCLK for the packet transmission and thus

Figure 3.3 A seamless stream of data from an open row in the burst mode.



with 90ns only slightly less than the 100ns with BL=4. On the other hand, all transmissions of consecutive packets would worsen significantly, since in most chips we could not produce a seamless stream of data with BL=1, because Eq. 3.1 on page 74 would be violated. Therefore, BL is usually 4, Eq. 3.1 on page 74 is fulfilled and moreover we have

$$\text{tRAS} \quad \leq \quad \text{tRCD} + \text{tCL} + \text{BL} \cdot \text{tCLK}, \tag{3.2}$$

which means that a read access keeps a row open for at least tRCD + tCL + BL · tCLK time and thus tRAS does not contribute to the latency calculations and is sometimes omitted from the timing notation. In Eqs. 3.3 and 3.4 on page 78 we extend this reasoning to multiple data rate memory chips like DDR, QDR.

3.1.1.2 Bandwidth

To improve the sustained bandwidth, we can stream a whole row without intermediate latencies from a memory chip (Figure 3.3), although this may not be true for the memory systems (see Section 3.1.2 on page 79). But if we want to stream a larger piece of memory which occupies several rows, then the time intensive row miss from open mode would occur. We could avoid this by having several rows open simultaneously and some internal management would schedule the opening and closing of rows such that no additional delays occurred. There are memory chips which work in this way (Rambus [Crisp, 1997]) but because of costs and simplicity the standard SDRAMs can have only one open row. Instead, the different internal *banks* of the memory chip offer a cheaper solution to the problem called ***bank interleaving***. Thereby, only those row misses from open produce latencies, which open a new row in the same bank. If a row in a different bank is requested, the latencies can be hidden by issuing sufficiently early a bank activation and read commands on the new row. Depending on how good the command pipeline in the memory chip can deal with the early commands the latency of row changing can be significantly reduced or even completely hidden. Then even large memory blocks can be streamed with only one initial latency, thus approaching the peak bandwidth.

Reaching the peak bandwidth is prevented by the fact that the *memory refresh* for DRAMs requires each row to be updated every few milli-seconds to prevent the information from dissolving. This intervenes with the normal read and write operations, because the *memory refresh* uses the same logic, namely the *sense amps*. We have not discussed this in detail, because it is done automatically and we have no influence on it. The average empirical bandwidth loss due to refreshing is small (1-4%), but it can be crucial for real-time systems, since it is difficult to pin down the exact delays for a given application [Atanassov and Puschner, 2001].

We have so far discussed the read operation. Writing is very similar since after opening a row, we only need to write to the sense amps. When the row is closed the content will be written back to the correct row. Writing is also less critical because in most cases the processing elements (PEs) can issue several data packets for writing and do something else afterwards not caring about when the data will actually be written. This is impossible if they wait for the result of a read operation which will deliver currently required data for a computation. So while writing itself is fairly trouble-free the changes between reading and writing exhibit an additional latency, the ***turn around time***. This time is necessary to make sure that the operations do not interfere, i.e. that they complete before the other starts, so that it is clear whether the data at the data pins should be read or written. Recapitulating, we can say that it is impossible to reach the peak bandwidth even on the memory chip level, but we can come very close to it, if all addresses are known well in advance and can be issued early, so that the inevitable latencies of the data access can be hidden.

Now we turn our attention to some techniques which increase bandwidth in general, but unfortunately hardly improve latency. We have three options to accomplish this:

- Increasing the rate of data packets per clock cycle.

  So far we have only talked about Single Data Rate (SDR) memory chips, i.e. chips which can send or receive only one data packet per clock cycle. Double Data Rate (DDR) chips which send two data packets per clock cycle are already widespread, a few SRAM Quad Data Rate (QDR) chips also exist and eXtreme Data Rate (XDR) DRAM with even higher transfer rates per clock cycle are being developed. We should emphasize that in such designs only multiple data is sent per clock cycle, other information as control signals and addresses are not compressed in such a way. Therefore, the latencies of the chips remain the same, or even worsen due to the more complex designs, e.g. DDR SDRAM 10ns 3-3-3-8.

  The faster data transmission requires a modification of Eqs. 3.1 and 3.2 on the preceding page. Eq. 3.1 now becomes

  $$\text{tCAS} + \text{tCP} \quad \leq \quad \text{BL} \cdot \text{tCLK}/\text{DR}, \tag{3.3}$$

  where the Data Rate (DR) is 2 for DDR and 4 for QDR. So the burst length (BL) is sometimes higher to allow more time for the compensation of tCAS and tCP. In a QDR chip with BL=4, for example, the burst would need only one clock cycle, so that we would have to be able to issue read commands on every clock cycle to avoid gaps in the data transmission from an open row. This is seldomly possible, so that the gaps in the stream would destroy the bandwidth advantage of the quadrupled data transport. Eq. 3.2 is also adapted to

$$\text{tRAS} \;\leq\; \text{tRCD} + \text{tCL} + \text{BL} \cdot \text{tCLK}/\text{DR}. \qquad (3.4)$$

Because of the shorter data transmission, the above inequality is sometimes violated by multiple data rate chips. In a DDR 2-2-3-8 chip with BL=4, for example, a single read operation will keep the row open for $\text{tRCD}(2) + \text{tCL}(2) + \text{BL}(4) \cdot \text{tCLK}/\text{DR}(2) = 7\,\text{tCLK}$, but the specification requires it to stay open for $\text{tRAS} = 8\,\text{tCLK}$. So although DDR doubles the bandwidth additional latencies may occur, decreasing performance in codes with many random accesses.

The reason for pursuing the multiple data rate technology is its fairly easy and inexpensive fabrication from the SDR components. Because in a DDR all control signals operate with the normal clock cycle, we can simply use the normal memory core from a SDR chip and add a small buffer and additional circuits which multiplex two data packets, so that they are sent on the rising and falling edge of the clock signal. This means that we need a ***prefetch size*** of 2, i.e we retrieve twice the number of bits from the memory array per clock cycle. Since the general design of the memory array remains the same and only a very small part of the chip must operate twice as fast, DDR chips have become very popular in increasing bandwidth. In fact, the peak bandwidth really doubles, but in everyday applications, frequent random accesses make the performance gains far less significant. For our PDE solvers with a regular data-flow, however, the economic DDR chips are first choice in hardware architectures, and graphics cards use them and the optimized Graphics DDR (GDDR) variant almost exclusively.

- Widening the data bus of the memory chip.

  As explained in the beginning a typical SDRAM memory chip has a depth of 8 bit. Adding more pins is expensive, but to achieve high bandwidth for graphics cards, for example, the expenses are incurred and 32 bit deep chips are used. A complementing option is the widening of the data bus on the level of the memory system by lining up several memory chips (Section 3.1.2 on the facing page).

- Decreasing the clock cycle time (tCLK) of the memory chip.

  This seems to be the simplest and most efficient option. Unlike the multiple data rate designs we make everything faster, so that also the latencies decrease. Unfortunately, depending on the voltage and the density of the memory cell arrangement there are physical limits to how fast the column and row address signals and the pre-charge can be made and how long a row must stay open before it can be closed again. So while the frequency has been increasing, the absolute latencies hardly improved, which means that they worsened in multiples of clock cycles. Moreover, increasing the frequency of the whole memory array is a problem in itself, because as with other integrated circuits, higher frequencies mean higher power consumption and heat development. So instead, one reverts to the increasing of the prefetch size as in the case of multiple data rate chips.

> The memory array of a DDR-400 memory chip operates at a 200MHz clock speed, but data is output at 400MHz because of the prefetch size 2. As it is costly to further increase the speed of the core, the DDR2-400 chip features a 100MHz memory array with a prefetch size 4 and data output at 400MHz. The output logic operates at the clock speed of 200MHz and multiplexes the four data packets onto two clock cycles, such that the data output is again at 400MHz. The reason for introducing this Second Generation DDR (DDR2) standard are mainly economic costs. The 100MHz memory array is cheaper in production and increasing the frequency of the core to 200MHz easily doubles the bandwidth again, while increasing the frequency from 200MHz to 400MHz in standard First Generation DDR (DDR1) is more costly. Nevertheless, for the high-end segment both techniques are combined in ultra-fast Third Generation GDDR (GDDR3) 1.25ns chips which have the memory core running at 400MHz delivering data at 1600MHz or 1.6Gb per pin.
>
> The difference between the DDR2 and the QDR technology is that a DDR2 chip outputs the four prefetched packets on two clock cycles, while a QDR chip would do this in one. So the handling of QDR signals requires more precisely timed logic, since two of the data packets are not synchronous to the clock signal edges.

Recapitulating, we see that the exponential increase in bandwidth is based only in parts on frequency increase. In 1990 both the i486 and the memory ran at 33MHz [Risley, 2001], nowadays the Pentium 4 processor [Intel, 2004b] is at 3GHz while the memory array of a standard DDR2-400 chip runs with 100MHz and the currently fastest GDDR cores operate at 500MHz. So even with the DDR technology the growth of the bandwidth of the memory chips lags behind. Additional arrangements on the system level, discussed in the following section, help to close this gap to a certain extent. Latencies have improved only slowly from around 100ns in 1990 to 50ns (25ns high-end) for a random access nowadays. In many places of this section we have therefore emphasized how important a regular or at least predictable data-flow is, which allows to hide these latencies.

3.1.2 Memory System

The processing elements (PEs) in most computer systems communicate with a memory controller which handles the main memory over a data bus and an address bus. In most microprocessors the bandwidth of the Front Side Bus (FSB) determines how fast the processor can read or write data. The bandwidth of the memory system data bus does not have to equal the bandwidth of the FSB, although this is desirable. Currently, the standard data bus width in PCs is 64 or 128 bit, in graphics cards 128 or 256. During a read or write operation all lines of the bus are used simultaneously and their content defines a bus word. Because a typical SDRAM memory chip has a depth of only 8 bit, eight of them are arranged on a ***memory module*** to provide the 64 bit of a word. Similar eight 32 bit deep GDDR chips sum up to the high-end 256 bit wide graphics card buses. For the PC memory modules this means that physically the first byte of the address space lies in the first chip at (0,0) in the memory array, the second byte in the second chip at (0,0), . . . , the ninth byte in the first chip at (0,1), etc. When we open a row in a chip, we thus actually open the addressed row in all chips simultaneously, and the union of these rows is called a ***memory page***. Notice that for the data bus it does not matter how the page is arranged, whether in eight 8 bit deep chips or sixteen 4 bit deep ones. In the discussion about memory timings one therefore talks about an open or closed page, and page hits and misses, rather than referring to the rows.

The options for increasing the bandwidth on the system level are similar to the ones available for the memory chips:

- Increasing the rate of data packets per clock cycle.
  Similar to the DDR design of memory chips there are also plans to multiplex data for the same clock cycle on a memory module basis. Thereby, two memory modules are stick together and are clocked with slightly displaced clock signals, such that their output can be multiplexed delivering twice the number of data per clock cycle. The modules could even themselves contain DDR chips, so that the output would be Quad Data Rate (QDR).

- Widening the data bus.
  This has already happened several times in the past. Doubling of the data bus width immediately doubles the peak bandwidth, but programs can only benefit from it if on average most of the bytes contained in the transported word are really utilized. Otherwise, a lot of memory is moved although only a very small part of it is really needed. So for applications which have a regular access pattern to large memory blocks the gains of a widened bus are enormous, and architectures such as graphics cards, which are designed for the processing of large data chunks have a 2-4 times wider bus than the common PC.

  On the mainboards of PCs the data bus can also be widened to 128 bit by using two different memory modules as accumulative 64 bit channels, instead of widening the memory modules themselves. This has the advantage that one can still utilize the same memory modules as for other single channel mainboards.

- Decreasing the clock cycle time (tCLK) of the data bus.
  This puts requirements on the quality of the mainboard and the memory controller, just as the production of equally fast memory chips requires special care. Therefore, high clock cycle buses are more common to graphics cards where the interaction with the bus is more restricted and bandwidth is often the bottleneck, than to PCs where also other hardware components interact with the bus and latency is the bottleneck.

As the memory controller must handle memory requests from all hardware components, the main PEs which should perform our computation do not always get full attention. If the access rights on the bus are handled in a naive way, e.g. but rotating them among the devices, then the sustained bandwidth for the PEs decreases even when no other requests are pending. Only in newer PCs the processor can obtain an uninterrupted stream of data from the memory as provided by the burst mode of the memory chips (Figure 3.3 on page 76).

In the previous section we have seen that the clock frequency of memory chips has grown much slower than that of micro-processors. The reason why DDR2-400 chips with a 100MHz memory core and 8 bit depth can nonetheless provide enough bandwidth for the 6.4GB/s demand of the Pentium 4 QDR FSB at 200MHz, are additional techniques which increase the bandwidth, namely: the prefetch size 4 with multiplexed output, grouping of memory chips to form 64 bit deep memory modules and using of two modules as separate channels resulting in a 128 bit wide bus. Similar 32 bit deep DDR-950 lined up to a 256 bit wide bus offer currently the highest peak bandwidth of 30.4GB/s to feed the GeForceFX 5950 Ultra [NVIDIA, 2004] running at 475MHz.

By sending data from the processor to the memory controller and further to the memory modules additional latencies occur. Similar to the other latencies, this is not critical if a sequential data stream is requested, because then the system latencies occur only once in the beginning, but it makes random accesses even more devastating in terms of performance.

3.1.3 Memory Hierarchy

Bandwidth can be increased and latencies reduced substantially if memory resides closer to the PEs, ideally on the same ***die***, i.e. on the same piece of silicon on which the processor has been fabricated. It is however very expensive to put large amounts of memory on the same die as the PEs. Instead, small blocks of local memory store accessed data and intermediate results for fast reuse. The local memory is named ***cache*** if it is transparent to the PEs, i.e. the PEs still use global addresses and the cache determines on its own if it already contains the requested data and delivers it instead of an access to the main memory.

Micro-processors tend to use caches, while reconfigurable hardware usually exposes the local memory directly to the programmer. Depending on how close the memory lies to the PEs, we speak of ***registers*** (L0) and L1, L2 or L3 memory/caches. In nowadays common micro-processors registers come in numbers of tens to hundreds and sizes of 32 bit to 128 bit. Sizes of L1 caches go up to hundreds of KiB, for L2 up to several MiB, and for L3 up to tens of MiB. While the registers and the L1 cache are on-die with the processor, L2 is usually on-die or at least in the processor package, while L3 is less common and may be placed similar to L2 or outside of the chip.

The main advantage of the close integration is the opportunity to run the caches at the high processor clock rather than the much slower memory bus clock. Moreover, caches are built from SRAM and exert far less latencies than the memory bus with DRAM. The L1 cache has often only 2-3 clock cycle latency, the L2 cache 5-10, for the L3 cache it is usually more than twice the L2 cache latency, but still less than the random access over the memory bus which takes 100-200 clock cycles. Notice that here we talk about latencies as multiples of the processor clock cycle and not the much slower memory bus clock cycle. Bandwidth also decreases away from the processor core, whereby there are again significant differences between the theoretic peak bandwidth and the sustained bandwidth of different memory access modes. The different speeds of the caches lead to memory access performance graphs staggered by the size of the caches. Consult [Gavrichenkov, 2003] for an evaluation of such synthetic memory benchmarks for the current high-end PC processors Pentium 4 [Intel, 2004b] and Athlon 64 [AMD, 2004].

Caches turned out to be a very effective way of reducing memory latencies and increasing the bandwidth to the PEs. Empirical analysis for common PC applications shows that the cache system has over 98% hit rate, i.e. in less than 2% of the cases data must be fetched from the main memory. However, this fact suggests a too high performance gain. Recalling the general arrangement of memory cells from Section 3.1.1 on page 71, we understand that it is easiest to organize access to caches in whole lines. A typical L1 cache line is 64B wide, L2 64-128B. This means that when we request one byte, we get 64 of them into the cache. This is done, because we are likely to use some of the other bytes in the following. But if we do not, this does not necessarily mean getting a bad hit rate. We could be repeatedly requesting several of such isolated bytes each from its own cache line and getting a good hit rate, although we never use any of the other bytes in the cache lines. For performance it is therefore important how predictable the access to a new memory address is. Only if the address is known well in advance than the fetching of a whole cache line does not incur performance, because it does not stall the current computation which takes place simultaneous to the data transmission.

The latencies and hit rates given above depend on the ***cache associativity***. In an N-associative cache one has to check the tags of N cache lines to determine whether a given address in the main memory is cached, where the tags specify the main memory line which is mapped to the cache line. The smaller N is the less comparisons against the tags must be done, decreasing the latency. But the larger N is the greater the number of cache lines which may be used for the mirroring of a given address, increasing the probability of cache hits. The extreme cases are a directly mapped cache (1 associativity), which associates each address with exactly one cache line and thus needs only one tag comparison but has a lot of memory competing for the same line, and the fully associative cache (all cache lines associativity), which can map any main memory line to any of the cache lines and thus needs to check all tags but can always involve all lines in the caching. The 4- or 8-associative caches are common and form a good compromise between short latencies and high hit rate.

Let us summarize the data access handling by following the request from the processor for a single bit which resides in the main memory. First, we have a latency incurred by the L1 cache miss and the L1 cache requests 64B from the L2 cache line. The L2 cache also suffers a miss and requests the 64B from the memory system. After additional memory system latencies the memory controller opens the requested page. This takes some time as it means measuring and amplifying the charge of the capacitors in the corresponding row. Then a series of read commands providing the column addresses starts the transmission of the cache line in burst mode. A few more cycles pass before data from the memory chips traveling through the mainboard reaches the processor. The processor stores the incoming data in the caches, and on arrival of the requested integer, passes it to the PEs for the one bit check. Certainly, this is an extreme example, but it illustrates the complexity of a memory access. Moreover, even when a stream of instructions is available to the processor a cache miss is still devastating and the techniques to handle them while several instruction threads are executed in parallel are clearly not trivial [Karkhanis and Smith, 2002].

To clarify the bandwidth mismatch between micro-processors requirements and the memory system performance we continue the example from the previous section. For a Pentium 4 with a FSB800 we have seen that the dual channel DDR2-400 modules can provide the 6.4GB bandwidth for the Front Side Bus (FSB). But the processor itself running at 3GHz can process four 32-bit floats in one clock cycle using the Streaming SIMD Extensions (SSE) and would thus require 48GB input bandwidth per float, which is 7.5 times higher than the FSB bandwidth. If we consider a single additions then two operands would have to be retrieved and one result written in each clock cycle resulting in 22.5 fold mismatch. Factors like these are typical for the disparity between the required peak processor and the peak memory system bandwidth. Moreover, we have only considered the SSE unit, although a micro-processor also contains several additional Arithmetic and Logic Units (ALUs) which can run in parallel. Therefore, the L2 cache in the Pentium 4 3GHz provides 96GB peak bandwidth, so that it makes a lot of sense to load all data which will be needed in later calculations into the cache lines in advance, this is called a ***cache prefetch***. Compilers should provide such hints to the processor, however, in programs which use a lot of data, such that the L2 cache cannot hold all of it, it is in general difficult to decide at compile time which data should be prefetched, in particular, as this depends on the various memory timing parameters of the system. The processor itself issues prefetch instructions at runtime by speculating which code will be executed next and such optimization of the memory behavior can be fairly successful when performed carefully [Hu et al., 2002]. But again, for optimal performance we are much better off with a regular data-flow.

In contrast to micro-processors the bandwidth requirements of a GPU and the provided memory bandwidth on a graphics card used to be better balanced. The GeForceFX 5950 Ultra running at 475MHz, for example, can process 8 texture values in one clock cycle and each texture value may consist of up to four 32 bit deep colors. So if we want to keep all PEs busy all the time, we have a memory bandwidth requirement of $475\text{MHz} \cdot 8 \cdot 4 \cdot 32\text{b} = 60.8$GB/s. The memory provides half the bandwidth, so the mismatch is much smaller. But with the new generation a change occurred, as the GeForce 6800 Ultra has 4 times as many PEs but only slightly higher memory bandwidth, resulting in 5.8 bandwidth mismatch factor, more similar to the micro-processors. This tendency is likely to continue because computer games now also use configurations with higher *computational intensity* and the integration of additional PEs into the GPUs is cheaper than the corresponding bandwidth increases. What is more important in favor of GPUs is the processing of the data in data streams, which minimizes latencies and maximizes throughput (Section 3.2.4 on page 93).

In the second part we will turn our attention to the computation itself. Once the data arrived at the PEs, the question arises which organization of them will deliver the result fastest. After having experienced the relevance of the data access pattern on the overall performance, the computing architectures will also be examined with respect to latency and bandwidth efficient memory usage.

3.2 Computation

In general we have talked about processing elements (PEs) so far, because the hardware elements which perform the computation can be very different. However, sometimes we referred to micro-processors which serve as Central Processor Units (CPUs) in our PCs, because this implicitly defined a computing paradigm known as ***von Neumann*** or ***instruction-stream-based (ISB)*** computation. An ISB processor executes an instruction stream and reads and writes data according to the instructions. This is the predominant computing model, not necessarily because it is the best for all applications, but rather because of historic and economic considerations. We have already seen for the memory chip design that economy has often a stronger impact on the design of computers than performance. So only the ever growing size of multimedia data together with the *memory gap* problem made way for the dichotomic ***anti machine*** paradigm from an economic point of view. In the following we adopt the terminology for the classification of the architectures from [Hartenstein, 2003].

The anti machine paradigm is based upon ***data-stream-based (DSB)*** computations, i.e. the PEs in a DSB processor are configured to perform a certain computation triggered by the arrival of data at their inputs. Obviously, the order of the data plays a crucial role in the evaluation. The data streams are generated by a ***data sequencer*** programmed with ***flowware***, just as an ***instruction sequencer***, which is usually a part of the ISB processor, generates an instruction stream determined by software. Different operations are realized by reconfiguring the PEs to a new state and streaming the data through them again. For this to happen the PEs must be reconfigurable, in which case we speak of ***morphware***, i.e. reconfigurable hardware, in contrast to hardwired and non-configurable hardware. The code which is used to reconfigure morphware is named ***configware***, Table 3.2 on the facing page summarizes the relations in the nomenclature.

However, the classification from the table suggests a false homogeneity within the classes. The design within each class may vary significantly leading to optimal performance in different application areas. We can only scratch the surface of the different computing paradigms and refer to [Becker, 2002; Bell, 2000; Hartenstein, 2003; Herz et al., 2002] for further reading and references on this topic. But before looking at the two computing paradigms in more detail we must first examine how to measure performance and how the general concept of parallelization can help to increase it.

3.2.1 Performance

Similar to the memory characteristics where we have considered the performance attributes of latency and bandwidth (Section 3.1 on page 71), for the computation we also have a two-fold measure of ***throughput***, the amount of data that can be processed per second, and ***latency*** or execution time, the delay from the computation's beginning to its end. Both measures depend on the problem, but allow a comparison of different architectures. Additionally, measuring operations per second (OPS) may be useful to compare different solvers on the same machine

Table 3.2 Sources, execution units and platforms of instruction-stream-based (ISB) and data-stream-based (DSB) processors. The ISB processor usually contains the instruction sequencer, so that software defines an instruction stream which determines both the instruction scheduling and the execution in the PEs. Similarly, a DSB processor may also contain the data sequencer, so that a common configuration determines both the data scheduling and the execution in the PEs. But the conceptual distinction between flowware and configware is kept up, as it is reflected in the co-design of two different parts of the configuration.
Hybrid designs mixing the different execution units also exist. They may involve the co-design of all software, flowware and configware.

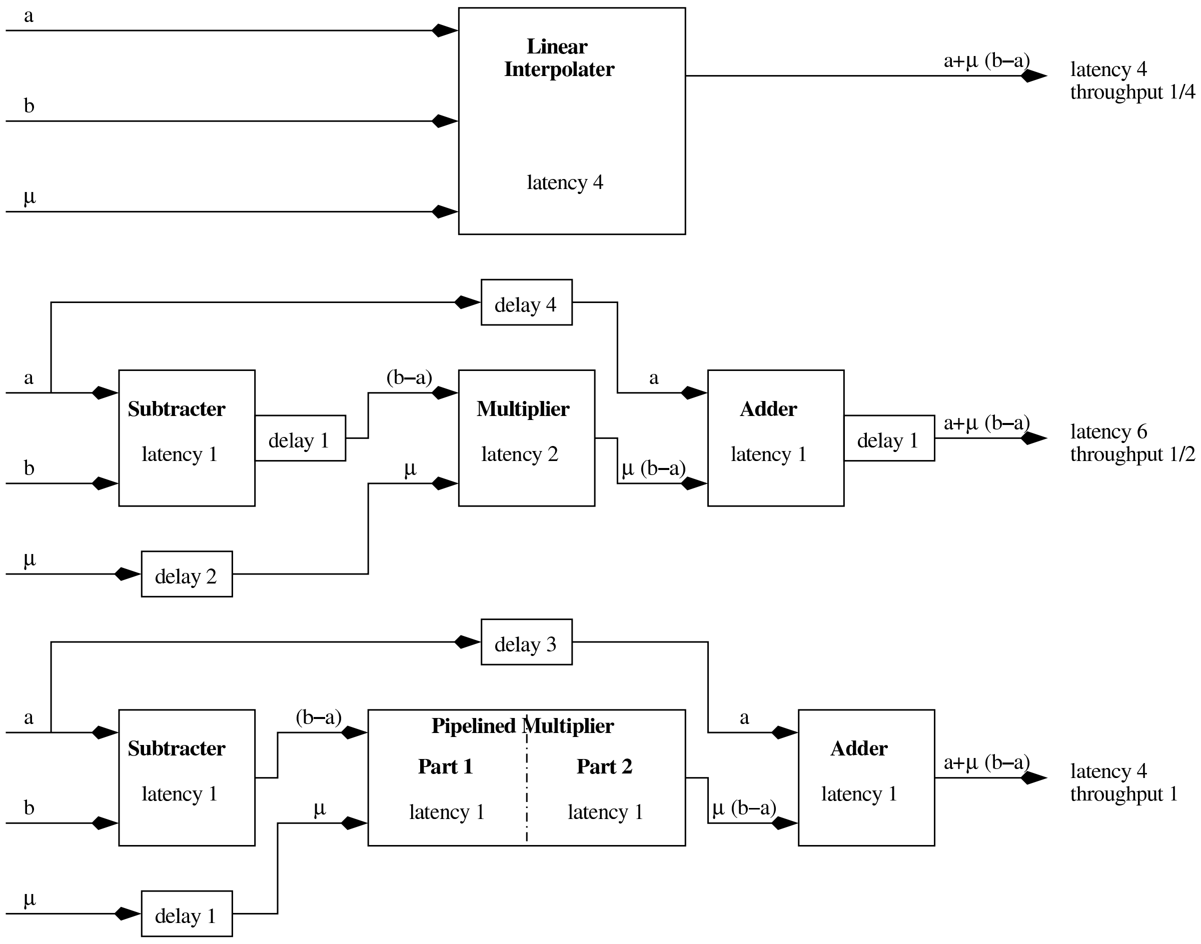
source	execution unit	platform
software	instruction sequencer	hardware
software	PEs in the ISB processor	hardware
flowware	data sequencer	hardware/morphware
configware	PEs in the DSB processor	morphware

and against the maximal OPS performance of the architecture. However, OPS numbers are less useful in comparing different processors, because the general term 'operation' may imply different amount of actual processing. The common measure FLOPS for example, ignores the fact that the transistor count for a multiplier grows quadratically with the data width of the operands, while the transistor count for an adder grows only linearly, thus 10 FLOPS (multiplications) is much more powerful than 10 FLOPS (additions) in reconfigurable hardware.

In general, we can say that latency is more important for the processing of small data packets, while throughput is crucial for large data blocks. Latency is often significant for applications like emergency signals, system surveillance, real time process control, financial orders, search of data base entries, etc. Throughput is the decisive factor in our case of large images and in many other scientific computations operating on huge data blocks.

In the design of hardware architectures or in reconfigurable systems throughput can be often traded for latency performance by a technique named ***pipelining***. Thereby, several PEs are lined up to perform a complex operation (Figure 3.4 on the following page). Each of them has the same latency, enabling a synchronized hand in hand execution. This has the advantage that the first PE does not have to wait for the last PE to complete, before starting a new computation. Instead, as long as the input is available all PEs keep on computing and pass the intermediate result to the next one. If the pipeline is long, the overall latency may sum up to a large value, but the throughput only depends on the common latency of the individual PEs. Obviously, less complex PEs have smaller latencies, so that the pipeline grows but the throughput increases. Longer pipelines also diminish the problem of ***hot spots***, i.e. particularly high power emergence on a small area of the die, and thus allow to further increase the frequency of the design. Indeed, this is a rule of thumb: longer pipelines enable higher clock frequencies.

Figure 3.4 An example of throughput increase through pipelining. Three different designs show the implementation of a linear interpolation $a + \mu(b - a)$. The first uses an integrated circuit for this operation. The second is a pipeline arranged of components with latency 2. The overall latency worsens, but the throughput is doubled. The third design shows an optimized pipeline with a quadrupled throughput in comparison to the initial design.



Let us elaborate on the pipelining example from Figure 3.4. An operation like linear interpolation $a + \mu(b - a)$ which takes say 4 cycles to complete in a certain integrated circuit, could be split into two adders with 1 cycle latency each and a multiplier with 2 cycles latency, doubling the throughput to 1 datum per 2 cycles. The pipeline requires more resources than the integrated circuit, because it cannot reuse one of the adders for both operations and more control logic is needed. But since the multiplier consumes most of the area, the percentage increase in transistors would be moderate.

The optimization of the pipeline continues in the third design. The additions have a smaller latency than the multiplication and are delayed in the pipeline to fit the common speed. But the multiplier can also be a pipelined design with sub-circuits' latencies matching the unit latency of the adders. Since the new maximal latency of the components in the pipeline is 1, we have doubled the throughput again to 1 datum per cycle.

We see that pipelines can be optimized in detail if we have the possibility of splitting and merging of the functional units for an optimal timing. In this way morphware often allows to multiply the throughput of the initial implementation. In the hardwired case only general and not application specific optimizations can be applied.

3.2.2 Parallelization

If we have only one elementary PE there is not much to optimize concerning the computation itself. Similar to the memory design we have the options of:

- Increasing the frequency.
  this has been done very successfully over the years, but at some point there is always a limit and too aggressive frequency settings may impede the use of the other optimizations.

- Widening the data path.
  this has also been done to some extent, but computing everything in 64-bit is not really a performance gain, if all numbers lie well within 32-bit. We might make use of the other 32-bit by packing numbers together. This, however, will probably mass up the code or even decrease performance due to frequent packing and unpacking operations. Therefore, the hardware itself often offers to use wide ALUs as two parallel ALUs with halved data width.

- Increasing the functionality of the PE.
  We construct a PE which will process more complex commands, which otherwise would have needed several simple commands and more time to execute, e.g. we could have a multiply-accumulate (MAC) instruction performing a multiplication and addition in one cycle. However, within such a PE there must be a multiplier and an adder, so why not speak of two concatenated primitive PEs. The question is whether this separate logic circuits can be used individually. Therefore, we restrict the use of the term *PEs* to specify the smallest individually controllable processing elements. So we could realize the MAC operation by two elementary PEs and this opens up the extensive topic of parallelization, the idea of arranging several PEs to accelerate the computation.

The main reason for considering parallelization are economic costs. Of course, parallelization is concerned with performance, programmability, reliability, power consumption and flexibility, i.e. the efficiency in adapting to different application requirements, which includes features like mobility, security, maintenability and scalability (the efficiency of increasing performance by adding more computing units to the system). But ultimately this all boils down to optimizing costs under the restrictions of the envisaged application area. We have to make this point clear because from a scientific point of view it would be more agreeable to think of a resource optimization under certain inherent constraints, but actually it is cost optimization and we cannot exclude the impact of the external factors of the market, which sometimes render solutions with a poor utilization of resources cost efficient, e.g. due to cheap availability of these resources through mass production.

Similar to the hierarchy of caches parallelization can take place at various levels and the closer the parallel units lie together the better the connection between them and thus the less delay in information exchange occurs. There are numerous levels of parallelization and for clarity we order them into a strict hierarchy omitting minor branches:

- Logic circuits in a PE.
  Even the smallest PE may exhibit parallelism either by *pipelining* or simply multiple inputs and outputs. On this level the parallelism is usually not directly accessible to the programmer, because independent control of the individual circuits requires a huge instruction or configuration stream and a lot of interconnect.

- PEs in a core.
  Most large scale hardware design projects do not start anew at the level of PEs. The design itself is hierarchical and uses cores as predefined assemblies of PEs of various complexity to arrange a more complex design. A major advantage is the availability of Intellectual Property (IP)-cores with exactly specified functionality and behavior from various vendors. The opportunities for parallelization in a core are numerous. At this level the driving cost factor corresponds to a performance per transistor ratio. Thereby both the absolute reduction of silicon and decreased power dissipation are the main benefits.

- Cores on a die.
  Some stand-alone processors on the market are also available as IP-cores, so that with additional communication logic one can quickly integrate several processors on a die. With modern fabrication process even embedded DRAM, analog circuits and opto-electro-mechanical systems can be integrated in a System-on-a-Chip (SoC) on the same die. Frequently reconfigurable hardware is also added to form a Configurable System-on-a-Chip (CSoC).
  The Simultaneous Multi-Threading (SMT) is a contemporary strategy for parallelization, where a single core processor pretends that it contains two cores by distributing the computations intelligently onto its parallel PEs.

- Dies in a chip package.
  The reason for having several dies in a chip package rather than putting it all on one die, lies partly in the silicium processing: memory and PEs have optimized processes of their own and smaller dies are less often corrupted by fabrication errors. Also, the individual dies may be produced in masses even though the concrete assemblies are needed in much smaller numbers. Therefore, from here on overall area efficiency is not the main cost criteria anymore, since mass production of standard components may render the assembly of them cheaper than the design of a new, more area efficient problem specific SoC.

- Chips on a card.

  Several processor and memory chips may be closely interconnected on a card, although in a common PC the chips are usually placed directly on the board. The assembly on a card is similar to the assembly of dies in a chip package but on a higher level. It makes sense mainly for Massively Parallel Processing (MPP) systems to introduce another level of interconnect proximity, because the larger the distance between the processors becomes the slower is their connection.

- Cards on a board.

  Several cards are put on a board. Additionally it includes memory and hardware components for interfaces to a network, mass storage devices, extension cards, etc. The typical example is the mainboard in a PC.

- Boards/racks in a computer.

  While a PC usually contains only one board, there is a whole hierarchy of board integration in High Performance Computing (HPC) ranging from multi-board arrangements to meter high racks which in multitude form a supercomputer.

- Computers in a cluster.

  Two to thousands of computers can be arranged in clusters, making use of the network capabilities of the computers for the interconnect. Such clusters may be build up from various categories of computers ranging from simple PCs to HPC machines. Usually cheap standard PCs are used and only the interconnect is of higher quality, as it often is the performance bottleneck in these inexpensive parallel computers.

  In this context the term ***system node*** refers to the smallest common building block of a computer system. In case of a cluster each of the computers forms a node, while for dedicated parallel computers the individual boards are the nodes.

- Clusters in the world.

  Distributed computing has become a major keyword in recent years. Indeed, the idea of joining computing resources around the world is intriguing. Both fast fiber connections between major HPC centers and slow networks between numerous PCs exist. Besides performance gains from parallelization, distributed computing addresses the very important property of system stability in large computer systems.

At all these levels the question arises what is the best topology for the communication between the computing units. Again, costs play an important role, but also the general assumption on how much communication will be required between the units and of which structure the data streams are. Common interconnect topologies are linear, bus, ring, star, tree, hypercube, mesh, and fully connected. Hybrid topologies also exist. This classification applies basically to the parallelization levels above the node level, because below we often do not need any direct communication at all. Instead, information is exchanged indirectly by using shared memory for the intermediate results. This shared memory may be single registers, register files, embedded RAM or local caches. Depending on how close this memory resides to the PEs different latencies will occur while exchanging the intermediate results between the PEs in this way. On higher levels a direct communication may turn out faster, e.g. some processors have an additional bus for the direct communication with other processors, which is much faster than the access to the shared memory on the node. It is clear that the distribution of memory and the choice of an appropriate interconnect topology play a crucial role for the performance of the parallel system. We distinguish the following basic types:

- ***Shared Memory***

  All nodes share the same global memory space. The memory is accessed over a common bus or a connection network. Caches embedded in todays micro-processors violate the totality of the sharing to some extent. Usually the processors themselves ensure that the contents of all caches is coherent. Non-cache-coherent architectures are easier to build and save on implicit synchronization but are very difficult to program.

- ***Distributed Memory***

  Usually on the board and card level, memory is distributed over the whole system and network requests to other boards must be issued if other than local data is required. The speed of the network is often a crucial parameter in distributed memory systems. In a Massively Parallel Processing (MPP) system, where several boards reside in a common rack close together, high speed networks can be used. For inexpensive clusters which connect standard PCs slower standard network components are more common.

- Non-Uniform Memory Access (NUMA) architecture

  This is a system in which the speed of an access to the logically shared memory varies with the actual physical location of the data. This is a common memory model for parallel arrangements with more than 8 nodes, because the costs of a symmetric access of all nodes to the memory grow quickly with their number. But more nodes also increase the complexity of the cache coherency problem. The logically shared memory space need not be shared physically. But NUMA alone usually refers to shared memory architectures or those with a fast hardware mechanism for retrieving data from different physical memory locations.

  In case of a truly distributed memory architecture which offers a logically shared memory space one speaks of Distributed Shared Memory (DSM). The sharing is achieved either transparently by specific hardware components or an operating system, or handled more explicitely by the user with an appropriate programming model. In either case the aim is to distribute the data efficiently with minimal traffic across large distances. In comparison to an explicit message passing between the nodes of a distributed memory architecture, the DSM model is more convenient for the programmer but may fall short of finding the optimal distribution and traffic of data among the nodes.

  Since NUMA only states a non-uniformity the performance of these architectures differs significantly depending on the relation of the access times to the memory address positions in the logically shared memory space. NUMA shared memory realized by software has typically an order of magnitude higher latency than the much more expensive hardware solutions.

On the low levels where PEs or blocks of them have to be connected we have the rough rule that the transistor count equals costs, thus increasing the performance by a high number of transistors only slightly, is not an option. By the ***speedup*** we define a problem dependent function which for given $p \in \mathbb{N}$ equals the ratio of time spent on solving the problem using one computational unit, to the time spent using p computational units. Ideally, we expect a linear speedup, i.e. the computation time is reduced proportional to the number of computational units. In rare cases super linear speedup is achievable due to factors related to the overall architecture. If we dealt with only one problem then we could optimize the arrangement of the PEs to obtain the best speedup. But for a wide range of applications it is more difficult to determine how to organize multiple PEs for optimal average speedup. Let us examine some cases:

- Simple computations on independent data entries.

  This is a situation with low *computational intensity* where one or very few operations are executed on each data entry. Then ***parallelism in breadth***, i.e. the arrangement of many PEs side by side, allows to process several data entries in parallel increasing speedup almost linearly. Usually all PEs can contribute to the computation, but the memory bandwidth requirement also grows linearly with each PE arranged in breadth order. Graphics hardware utilizes mainly parallelism in breadth.

- Complex computations on independent data entries.

  In this case many computations have to be performed on each data entry and we have a high *computational intensity*. ***Parallelism in depth***, i.e. the arrangement of PEs one after another in a pipeline, allows each PE to work on a small piece of the problem with the results of the preceeding PEs. The side by side arrangement would also allow a processing in parallel, but a lot of intermediate results would have to be written and read from memory for each step of the problem solver, thus unnecessarily increasing the memory bandwidth requirement. The pipeline, on the other hand, is very memory efficient, because ideally each datum has to be accessed only once. However, it is more difficult to utilize all of the available PEs for a given problem when arranged in depth order. If after building the pipeline some PEs are still available, but do not suffice to build a second one or to lower the common latency, it is not obvious what to use them for. If there are too few PEs to build the whole pipeline, then intermediate results must be transported to memory, but less often than in the side by side arrangement. Reconfigurable hardware often opts for the processing in deep pipelines.

- Computations on dependent data entries.

  Here one tries to exploit either of the above strategies looking for opportunities to compute some results independently of each other in a breadth arrangement and others in a pipeline arrangement. But if the data dependencies are changing constantly, then this approach requires a very fast dynamic interconnect between the PEs, otherwise the overhead associated with finding and configuring the parallelism will outweigh the advantage of it. Also, some codes are so inherently serial, that there is nothing that can be parallelized and thus no speedup can be gained from multiple PEs. ***Amdahl's Law*** specifies this observation: If s is the fraction of a calculation that is inherently serial, and thus cannot be parallelized, and $1-s$ the fraction that is parallelizable then the maximum possible speedup on p PEs is $\left(s+\frac{1-s}{p}\right)^{-1}$ and consequently limited by $1/s$ from above. Because of the danger of poor utilization of multiple PEs in serial codes, general purpose micro-processors contain only few parallel PEs. PE arrays with a fast dynamic interconnect offer more opportunities for parallel execution if an appropriate fast logic for the identification of the required parallelism exists [Mai et al., 2000; Sankaralingam et al., 2003].

We see that ideally the interconnect between the PEs is programmable, so that depending on the problem structure, the appropriate arrangement can be configured. In DSB processors this reconfigurability is often available, but in ISB processors the different arrangements are usually hardwired leading to a large variety of designs.

3.2.3 Instruction-Stream-Based Computing

Flynn's taxonomy [Flynn, 1972] classifies ISB architectures based on the number of streams of instructions and data:

- Single Instruction Single Data (SISD) - scalar

  This is the classical design used in a simple processor with one instruction stream operating on singular data packets. We also speak of a ***scalar processor***. In todays ***super-scalar processors*** the processor has several PEs and cares about the ***run-time scheduling***, i.e. the optimal distribution of the instructions onto the PEs for parallel execution during run-time.

- Multiple Instruction Single Data (MISD)

  A theoretic architecture, which would apply multiple instruction streams to a single stream of data. This could be suitable for problems where each datum undergoes a lot of computations.

- Single Instruction Multiple Data (SIMD) - vector, VLIW

  This is a very common approach to accelerate computations when large data blocks are processed, e.g. a ***vector processor*** applies operations to whole data vectors by a row of PEs. But also some common micro-processors contain SIMD PEs to exploit *parallelism in breadth*. The instructions in the stream can be either simple or fairly complex consisting of Very Long Instruction Words (VLIWs). The VLIW contains sub-instructions specifying different operation for the parallel PEs. In contrast to the super-scalar processor VLIW machines have to figure out the parallelism of instructions statically at compile-time and so we speak of ***compile-time scheduling***. The VLIW parallel execution is similar to the following MIMD model, but here we have still one instruction stream and the sub-instructions cannot be issued individually.

- Multiple Instruction Multiple Data (MIMD)

  Independent PEs can be programmed to perform different tasks on different data packets in parallel, if the computations are independent. The MIMD usually applies to the processor level, where each processor executes its own instruction stream, rather than individual PEs in one processor receiving the independent streams. Notice that the individual processors often work with the SIMD model. Because there are several instruction streams the task execution must be synchronized. This requires a network and message passing between the nodes. Within the node, which may contain several processors, other more direct communication is established.

The last class encompasses a huge variety of designs which differ in the number of the nodes, their complexity and the network topology. Also, additional distinctions about the relation of the instruction streams apply, e.g. the most common programming mode on several processors is Single Program Multiple Data (SPMD). Naturally, detailed classification of real systems is even more complex and hybrid architectures also exist [Duncan, 1990]. Often the programming interfaces rather than the details of hardware determine which sorts of parallelism can be really exploited [Rolfe, 2002].

Virtually all ISB architectures suffer from a memory bottleneck. The problem lies in the fact that the instruction stream prescribes both the consequent flow of instructions and the flow of data operands required by the current instructions. If the instruction stream changes unpredictably we get an unpredictable access pattern to the memory for the operands. But all of Section 3.1 on page 71 has made very clear, that it is crucial for the memory system to know the requested addresses in advance if high sustained bandwidth is desired. ISB processors fight the uncertainty in data access with speculative processing and memory prefetch, improving performance for inherently serial code, but not exploiting the full potential of highly parallel code. This seems appropriate for general purpose processors which mainly process serial code, but most processing time in scientific computations is spent in loops over large data blocks performing the same operation over and over again, and here massively parallel data-stream-based (DSB) execution is advantageous.

3.2.4 Data-Stream-Based Computing

DSB architectures are often classified with respect to the ***granularity***, i.e. the size and operand width of the smallest programmable units, the processing elements (PEs), and the arrangement of them. We distinguish three major flavors:

- Reconfigurable Logic (RL) [Bondalapati and Prasanna, 2002]

  FPGAs are the most important devices of the RL family and the only ones providing high logic capacities. We refer to [Brown and Rose, 1996] for an overview of the other RL devices. In the mid 1980s FPGAs began their success story, starting as small hardware simulation devices and leading to universally applicable multi-million gate chips of nowadays. The PEs of FPGAs are configurable n input 1 output lookup tables (LUTs), with a typical value of $n = 4$. By filling the entries of the LUT with values the desired logical function can be configured. Several LUTs are grouped together to Configurable Logic Blocks (CLBs) which usually also contain a register to facilitate the synchronization of the data-flow. The CLBs are organized in an array and the space between them is used for the interconnect network, i.e. configurable switches between data lines which allow the desired connection between the inputs and outputs of the CLBs. FPGAs are fine granular because the routing and processing of individual bitlines can be configured. Consequently, the *configware* for the FPGA consists of a large bit stream which contains the values for the LUTs and the routing of the data lines.

The advantage of FPGAs, the free configurability on the bit level, becomes a disadvantage when many standard ALUs are needed, which would consume far less transistors if hardwired or configured on the arithmetic level. Therefore, for the most common operations many FPGAs include hardwired units, ranging from multipliers for different operand widths to whole processor cores. Additionally, more and more RAM is available on the chip for the caching of input values or intermediate results. As sometimes even the largest FPGAs cannot hold the whole configuration for the problem solver, now the chips usually offer ***run-time reconfigurability***, i.e. intermediate results can be stored in memory and after reconfiguration of the FPGA the new configuration continues the computation. Some offer even ***partial reconfigurability***, which allows to reconfigure only parts of the FPGA, while others retain their configuration or even continue execution, which helps to hide the latency of the configuration process.

- Reconfigurable Computing (RC) [Hartenstein, 2001]

  RC refers to coarse-grain architectures with data line widths of 4 to 32 bit. The general arrangement is an array or line of tiles with an interconnect network similar to the FPGAs or crossbar switches. Simple tiles contain an ALU as a PE and may also have local registers and routing capabilities. In designs with complex tiles whole processors with local memory and communication resources are arranged. Naturally, architectures with simple tiles have many more of them and a more extensive network than the arrays of few complex tiles. Therefore, besides the different functionality of the tiles, the interconnection network differs strongly between the designs. Usually there is some nearest neighbor and some mid to long distance routing, but the number of the routing hierarchies and their complexity varies. Also the distribution and access to local memory is organized in different ways. Similar to FPGAs, RC architectures may contain local memory blocks and special computational units.

  In any case, the coarser structure in comparison to FPGAs needs far less configware, so that not only concurrent partial reconfigurability but sometimes even ***dynamic reconfigurability*** in RC systems is available, i.e. individual PEs are reconfigured and immediately start performing the new operation. In some architectures one can also adaptively reduce power consumption by switching off inactive PEs.

  If the predefined operations of the PEs are required by the problem solver, RC architectures have a better utilization of transistors than FPGAs, because the functionality is hardwired in parts. But if irregular bit manipulations require many ALUs or some unforseen other functionality must be awkwardly simulated, then the more flexible RL architectures have an advantage. But while RL architectures have become a mass market and apart from special purpose FPGAs their structure has strongly standardized, RC architectures still come in large variety of designs. So, often the development tools for RC systems lag behind. Moreover, there are a lot of hybrid architectures also including hardwired parts or even whole processors which require the integration of different programming models. Not seldom everything is even on the same die and we speak of a Configurable System-on-a-Chip (CSoC) [Becker, 2002].

- Stream processors [Rixner, 2002]
  Stream processors use multiple SIMD PEs to quickly operate on data streams. Internally the computations are not necessarily data driven but use for example VLIWs to trigger the operations. Nevertheless, the whole system is data-stream-based (DSB), because all components are focused on the generation and processing of data streams. A data sequencer, which may also be an ISB processor, generates the data streams which are fed to the PEs. In the Imagine stream processor [Kapasi et al., 2002, 2003] the instructions to the PEs are issued only once at the beginning of the stream processing, like a configuration, and than repeated from a local instruction cache. Other approaches propose to have an instruction stream corresponding to the data streams [Ulmann and Hoffmann, 2002].

  The Imagine project also balances the ratio of the different components, mainly memory and PEs, against their costs and power consumption [Khailany et al., 2003], avoiding the unbalanced situation in micro-processors where almost 50% of the die is consumed by caches. This leads to an architecture with a very high local throughput compared to the global bandwidth, most suitable for problems with high *computational intensity*. A memory hierarchy which allows the data reuse at different scales and programming models which encourage the programmer to expose these opportunities seek to reduce the demand for the narrow global bandwidth in data intensive applications. Also, the balanced power consumption of the system lends itself to scaling of such stream processing up to the level of supercomputers [Dally et al., 2003].

  Just like RC devices are basically specialized FPGAs, stream processor are in some sense specialized RC machines. A similar or even identical stream processing model is namely often used in various RC devices [Mai et al., 2000; Sankaralingam et al., 2003; Taylor et al., 2002]. By reducing the reconfigurability and focusing on the stream processing model alone, stream processors save in particular on the interconnect. Sop for the price of reduced flexibility they offer higher transistor efficiency for appropriate DSB applications.

DSB devices usually do not have problems with latencies because the data sequencer receives the requested memory addresses ahead of time and thus has enough time to retrieve the information and generate a continuous data stream. As DSB architectures often exploit the memory efficient *parallelism in depth*, their overall demand for data bandwidth is not necessarily higher than that of ISB processors. But the bandwidth may be too small if the memory hungry *parallelism in breadth* is required. For RL and RC chips this problem is usually solved by providing a lot of Input/Output (IO) pins, which allow the access to several memory chips simultaneously. The maximal bandwidth to external memory on large DSB devices can thus be raised to several GB per second. In practice, however, the problem is still present, because wide buses between the memory and the chip are expensive, so that the full potential bandwidth is often not provided on the boards. Luckily, internal RAM blocks can often be used as caches tailored exactly to the needs of the application, such that the external bandwidth requirements are further reduced. In stream processors the local memory cannot be used with absolute flexibility but the hierarchical memory structure is already arranged in such a way as to maximize data reuse on different levels.

3.2.5 Processor-in-Memory

We have repeatedly stressed the fact that overall system performance is hampered by the *memory gap*, the inability to transport the data from the memory to the PEs sufficiently fast. A radical solution to this problem is to bring the PEs directly to the memory array. This close combination of DRAM with PEs in the semiconductor fabrication has become possible only recently on a large scale.

While reconfigurable architectures and micro-processors have also embedded RAM, it is still clearly separated from the PEs. In Processor-in-Memory (PIM) architectures the coupling of the memory array with the PEs is much closer. Often a whole memory row can be processed in parallel by a collection of simple PEs similar to the reading or writing of a memory row. The gains in bandwidth and latency performance are tremendous and several ambitious projects focus on the development of the architectures and appropriate language tools to use them [Draper et al., 2002; Fraguela et al., 2003; Murphy and Kogge, 2000; Sterling and Zima, 2002]. The projects differ in the homogeneity of the systems, i.e. does a large number of PIM elements represent all computational resources or does it only accelerate typical parallel task beside a host processor; and in architectural innovation, i.e. is the conventional DRAM array augmented with processing capabilities for inexpensive and gradual introduction of the technology or is a completely new architecture with fine-grained interplay of PEs and memory blocks envisaged.

The design of completely new architectures offers so much freedom, that the question whether an ISB or DSB methodology should be used for the PIM elements is not that clear. The major problems of insufficient memory bandwidth and high latencies are solved by construction, and the challenges lie in an efficient communication between the fine-grained processing structures. Therefore, many of the PIM projects introduce new, memory friendly concepts of communication between the PEs, which use their own buses and run in parallel to the usual addressing of memory. But despite all the innovations, PIM architectures cannot make an irregular data-flow algorithm run fast. They rather push the idea of exploiting data coherency on the lowest level to the extreme, and algorithms which expose this parallelism to the architecture can be enormously accelerated. Although the utilization of the PIM concepts is fairly new and many question about the optimal arrangement and communication of memory and PEs are still open, the sample implementations and performance gains are very promising and commercial products will hopefully be soon available.

3.3 Hardware Architectures

In the previous sections we have discussed the concepts behind memory design and different computing approaches. Now we turn to some concrete architectures and examine how they relate to these concepts in general, and their efficiency for image processing applications in particular. At the end we dare to peek into the large scale future of hardware architectures and data processing.

3.3.1 Status Quo

Since mass production greatly reduces unit prices and significant initial investments are needed to benefit from latest semiconductor fabrication technology, one could expect the hardware market to be dominated by very few architectures. But on the contrary, it is quite diverse. The diversity stems from the applications' requirements. Even the best price-performance ratio becomes irrelevant if the corresponding device is not suitable for the envisaged application. Performance is of course an important factor, but flexibility, reliability and in recent years most notably power consumption often tip the scales. In practice the hardware must comply with all requirements to some extent, such that several distinct markets dedicated to specific weightings of the requirements can thrive simultaneously. So far we have always discussed hardware with a focus on large scale image processing, a task unattainable to mobile devices. Although we will keep this focus on PC based solutions, the reasoning about a regular data-flow with exploitation of data coherency serves also the purpose of power reduction, a key feature of embedded processors.

3.3.1.1 Micro-Processors

Micro-processors are ubiquitous as CPUs in our PCs. The performance of a single micro-processors, which has been increasing rather through higher frequency than per clock cycle functionality, is cheap in comparison to HPC machines. The exponential frequency increase requires ever longer pipelines in the processors to avoid dangerous *hot spots*. But the long pipelines together with the *von Neumann* computing paradigm have lead to the problem of performance destructive jumps in the instruction stream. The processor's effort to optimize and parallelize the execution of the instruction stream using multiple PEs thus requires a prediction mechanism for the conditional jumps.

The micro-processors have also dashed forward with their peak performance, such that the memory system cannot supply the required bandwidth for all the PEs and the latency for a random memory access in terms of processor clock cycles grows rapidly. Large caches and data prefetch strategies alleviate the problems but a considerable memory performance gap remains. The ISB computation worsens the problem in so far, as conditional execution of the instruction stream prohibits the early availability of later required memory addresses. So the prefetching depends strongly on a successful jump prediction.

The optimization techniques mentioned above are performed automatically by the micro-processors. Software may only provide hints for their application. This makes the programming model fairly care-free, but also wastes potential in cases where an optimal control of the task and data distribution could be prescribed explicitly. Two other processor extensions accommodate the programmer with more explicit control. Firstly, SSE PEs allow the use SIMD techniques to accelerate homogenous processing of data streams. Secondly, Simultaneous Multi-Threading (SMT) allows to hide a lot of the memory determined latencies and wait states through the interleaving execution of different instruction threads. The former technique offers an almost explicit access to the benefits of parallel computing, but depending on other pending commands some parts of the processing pipeline may be stalled screwing up the expected one clock cycle execution. In case of the SMT the actual interleaving strategy is totally up to the processor, which is probably right, because static analysis for multi-threading at the compiler level is very restricted.

In general, one may say that micro-processors offer a lot of performance for their price, though their resource utilization is fairly bad as compared to other architectures. The main advantage is the general purpose applicability paired with an easy programming model. The downside of this easiness is the very restricted influence on the efficient utilization of the processor's resources. The available optimization mechanisms require good knowledge of the internal processor structure and a lot of patience to be effective, because the actual internal processing model is very complex and not totally transparent to the programmer. This leads to the unfortunate situation that the ineffectiveness of some optimization attempts cannot be traced back. Moreover, very different optimizations may be required by different micro-processors depending on their complex inner arrangement.

Despite the problems in the optimization of instruction streams for the micro-processor mentioned above, there are some very successful demonstrations of the potential power of the processors. In particular the media and image processing applications we are interested allow on average speedup factors of 1.5 to 5 against unoptimized code [Abel et al., 1999; Franchetti and Püschel, 2003; Ranganathan et al., 1999]. The performance gains are mainly based on the exploitation of SIMD capabilities and the implicit parallel use of other PEs. The design of the Itanium(2) [Intel, 2004a] processor follows this observation and exposes more explicitely several parallel PEs to the programmer. In many cases, the compiler can parallelize the code accordingly on its own. But there is a need for appropriate HLLs and a hardware awareness to exploit such parallelism more thoroughly, and if this is not achieved by the highly parallel reconfigurable systems then hopefully the growing number of PEs in micro-processor will promote these languages to common use.

3.3.1.2 Parallel Computers

With parallel computers we refer here to systems which utilize a number of micro-processors for parallelization. Micro-processors alone are already difficult to optimize, but for parallel architectures the problem becomes really hard. From Section 3.2.3 on page 92 we known that parallel arrangement of computing units can occur at many levels. Together with the variety of the individual processors, interconnects and memory distributions there is huge number of different parallel systems. It is not possible to compile a single program to an efficient code on all of them. In the Message Passing Interface (MPI) [MPI committee, 2004] standard, for example, the programmer has still to consider the performance of the individual nodes, memory bandwidths at various levels, and the topology of the network to gain optimal performance. So the performance of parallel algorithms depends more severely than in the case of single micro-processors on the programmers knowledge about the specific system and the different parallel programming models [Leopold, 2000].

Nevertheless, parallel computers can have a good price performance ratio if the programmers take into account the features of the system. Especially clusters made up of standard PCs have gained high popularity in recent years [Baker/Ed., 2000]. They already occupy 7 of the top 10 places and more than 40% of the current (Nov. 2003) TOP 500 list of the fastest computers [TOP500 committee]. However, the LIN-PACK benchmark used for the ranking in the list reflects the performance for only one application scenario. Actually, it favors distributed memory systems, because the required communication is moderate. For some application classes clusters tend to be communication rather than computation bound. The site [TOP500 committee] lists also initiatives for a more application oriented benchmarking of HPC.

Most image processing tasks pose only moderate requirements on the communication between the PEs and have therefore been the target of many successful parallel implementations since the early days of parallel computing. Michael Duff gives a thorough overview on 'Thirty Years of Parallel Image Processing' [Duff, 2000]. Although more and more previously unattainable tasks can be handled with a PC, there are always more demanding image analysis and visualization tools and ever larger data sets. The implementations in the next chapter demonstrate that many data intensive applications can be solved in reasonable time with a PC and an appropriate Peripheral Component Interconnect (PCI) card, but there is a limit to this single component solution, which cannot render parallel systems obsolete. One can naturally think of parallel systems made up of graphics cards [Sud et al., 2002] or reconfigurable hardware [Nallatech; Star Bridge Systems Inc] rather than micro-processors. This architectures are evolving, but the challenge lies not only in the design of a high performance architecture, but also in the development of a programming model, which makes the whole processing power accessible.

3.3.1.3 DSP Processors

Digital Signal Processing (DSP) covers a wide range of audio/speech, image/video and sensor/control applications. Because of this diversity there has always been a number of Application Specific Standard Products (ASSPs) and dedicated Application Specific Integrated Circuits (ASICs) to serve the particular needs of applications. On the other hand, many DSP tasks have a lot in common even though the signals stem from different sources. On the low level even more similarities appear, like certain memory access patterns and arithmetic operations: multiply-accumulate, dot-product, scaling, biasing, saturation. DSP processors offer these capabilities together with SIMD PEs and an instruction set which has enough flexibility to implement different applications. Naturally, different combinations of the features result in different application stress and the number of available chips is enormous [Cravotta, 2003]. So even within a specific application area a choice among many DSP processors must be made. However, measuring performance among different chips is a challenge [Su et al., 2004]. In contrast to computing with a PC, embedded computing puts more emphasis on component integration, power consumption, reliability, predictable system behavior, programmability and costs rather than pure computing power.

The high-end general purpose micro-processors used to be faster than the strongest DSP processors [Bier, 2002], but they are fairly unsuitable for most embedded applications, because of high power consumption, difficulties in integration, unpredictable dynamic behavior (speculation, caches) and last but not least costs. Moreover, latest benchmarks on current DSP processor families [Su et al., 2004; Williston et al., 2004] show that for many applications the highly parallel architecture of DSP processors can now deliver comparable performance to the much larger micro-processors. The disadvantage of DSP processors is their lesser suitability for the implementation of the user interface and general operating system duties. Therefore, often both a micro-processor and a DSP processor are used together. If the processing requirements are still higher, then even more additional chips in form of coprocessors are needed. These may include reconfigurable hardware, ASSPs or ASICs. So DSP boards nowadays hold a variety of different chips and where the tools for the individual components leave room for improvement, a tool for the simultaneous co-design of hardware and software in such a polymorphous situation is still a challenge.

Concerning image processing we can say that in many embedded systems DSP processors still dominate, but the superior performance of highly parallel reconfigurable arrays has been widely recognized and put into practice [Tessier and Burleson, 2001]. A lot of commercial boards now offer a combination of DSP processors and FPGAs [Bondalapati and Prasanna, 2002], while first products with RC chips also exist [Becker et al., 2003]. While some Reconfigurable Logic (RL) and Reconfigurable Computing (RC) architectures claim to replace DSP processors altogether, which seems likely in the long run concerning the performance/power consumption ratio, the large DSP processor market is unlikely to disappear at once, also because the reconfigurable hardware requires a different programming approach, which still receives little attention by the computer science curricula [Hartenstein, 2003].

3.3.1.4 Reconfigurable Logic

We can characterize the computing models of ISB processors as ***temporal computing*** and of morphware as ***spatial computing***. In the first case the scheduling of the time ordered instructions onto the PEs is important, in the second the spatial arrangement and connection of them. Obviously the spatial model offers more room for parallelization [DeHon, 2002]. In an instruction stream we can use consecutive independent instructions for parallel execution, but this can be economically done only on a small scale. In a PE array only the amount of required communication between the PEs and the data bandwidth limit the parallelization. Massively parallel MIMD architectures form a kind of transition from one model to another.

Although the individual instruction streams still enforce temporal computing, the distribution of the tasks already requires spatial considerations. But in RL the spatial considerations go much further. Concerning a processor's performance we do not care how many transistors have been spent on an adder or multiplier. If we have several PEs in the processor than evaluating $ab - bc + d$ might be even faster than $(a - c)b + d$. But for RL the latter would be definitely the better choice because it involves only one multiplication and thus less resources. The free resources can then be used to implement another version of this computation, provided we need to process a lot of data in this way. This means that area efficiency translates directly into parallelism and further into performance gains in RL.

FPGAs are the dominant devices in the RL domain [Brown and Rose, 1996]. They have experienced a rapid growth both in size and volume. The mass production has also standardized the products so that now there are many commercially available boards and development tools [Compton and Hauck, 2002]. Different needs are covered with single FPGAs, multi-chip boards and hybrid architectures. Usually some processor performs the configuration of the FPGA, which is responsible for the fast highly parallel data processing. FPGAs have gained so much popularity because the development tools provided with the morphware have also matured. Compilation from High Level Languages (HLLs) like C and its hardware specific variants to a loadable configuration can now be done automatically. Even the programming of multi-chip boards is automated by introducing virtualized morphware resources [Enzler et al., 2003]. However, for optimal performance one often has to interfere with the automatic processes and hand-code at least some parts of the design in a HDL.

In image processing and other DSP applications FPGAs regularly outperform DSP- and microprocessors by an order of magnitude or even more, and this despite a lower clock frequency [Cerro-Prada and James-Roxby, 1998; Hammes et al., 2001]. The low clock frequency has even the advantage that less power is consumed. An analysis shows that FPGAs benefit mainly from the dense spatial parallelism [DeHon, 2000]. More beneficial factors are related to the DSB computing paradigm: efficient use of available IO by data streaming, application tailored data reuse, the flexibility of *parallelism in depth* or *in breadth* depending on the available data bandwidth [Guo et al., 2004]. Despite the efficient use of IO bandwidth, the increased number of parallel PEs as compared to an ISB processors may also lead to a memory bottleneck in FPGAs. For high speedups in RL, it is therefore crucial to have sufficient memory bandwidth and many problems scale directly proportional to this parameter [Benitez, 2003].

Beside the merits, there are also drawbacks to FPGAs. To allow the fine-grained configuration a lot of transistors are used as opposed to a hardwired design. If most of the Configurable Logic Blocks (CLBs) are used for the implementation of standard Arithmetic and Logic Units (ALUs) then this constitutes a wastage, and we would have been better off having the ALUs hardwired with programmability only at the function level. Especially the area consuming multipliers are therefore integrated as hardwired resources into new FPGAs, which partly alleviates this problem. The fine-grained architecture also requires a very fine interconnect on the chip, which consumes more area than the CLBs. If we assume that data is in general changed in bytes rather than bits, then a lot of logic for the interconnect can be saved. Additionally, the high level of programmability implies a large configuration file, and thus a comparably long configuration process. Introducing caches and compressed configurations cannot eliminate this problem in principle. The large configurations make FPGAs less suitable for *dynamic reconfigurability*. Finally, despite cycle accurate simulation tools, debugging of FPGAs configurations can be a real hassle given the enormous amount of configured lookup tables (LUTs). By sacrificing the bit-level configurability we can diminish the scope of these problems, but still retain the advantageous massive parallelism and memory friendly DSB processing. This brings us to the Reconfigurable Computing.

3.3.1.5 Reconfigurable Computing

In our rough distinction of RL and RC architectures we let RC start with a word length of 4 bit. Here, word refers to the smallest data packet which can be processed individually. Based on an empirical rule, one can derive that logic density, i.e. the fraction of area used for the PEs, is maximal for word lengths of 4-5 bit [Stansfield, 2002]. DSP applications have often even wider words and therefore many RC architectures use words of 16 or even 32 bit. If the architectural and the application word lengths are similar then unnecessary routing is minimized. But the word length is only one parameter to chose when designing RC machines. Many more options like complexity of the PEs, homogenous or heterogenous PEs, structure of the interconnect (local, global, hierarchical), positioning of local RAM elements produce a large variety of designs. An overview of academic RC architectures can be found in [Hartenstein, 2001], while commercial vendors are among others [Elixent; IPflex; PACT; picoChip; QuickSilver]. Computing machines based on fine grain RL are covered in [Bondalapati and Prasanna, 2002; Compton and Hauck, 2002].

For image processing where bit level operations are seldom required, but also for many DSP applications RC machines tend to be more suitable than the RL devices [Sueyoshi and Iida, 2002]. They form a happy middle between the fine grain configurability of RL and the high level programmability of DSP processors. The difficulties concerning RC are associated less with the hardware but rather with the unfamiliarity of many DSP developers with the DSB programming in form of configware and flowware, and the immature status of development tools. The situation is not totally comparable to the early days of FPGAs, because they could build upon the design tools used for ASICs. Those use Hardware Description Languages (HDLs) to describe the configuration, which are also little accessible to DSP developers, but meanwhile both the low level and the high level design tools have seen a tremendous development allowing access from HLLs to the processing power of FPGAs. This was surely a crucial step in the migration process of many DSP applications onto FPGAs. However, even these tools do not match the maturity of development suits on DSP processors, and therefore co-design with either hardware components are very common. Similarly RC devices are intended for use as coprocessors aside a DSP processor either on a common board or in a Configurable System-on-a-Chip (CSoC) and the availability of advanced programming tools for such compositions will be crucial to the success of the new devices. The implementations of RC in the next chapter makes use of the preliminary simulation and visualization tool of PACT [PACT]. Programming on RC architectures has the tremendous advantage over RL that after placing and routing of the algorithm, one can still identify the individual components in the configuration, such that debugging is a lot easier. With a visualization tool one can watch the cycle exact simulation and find the clock cycle and ALU where the computation goes wrong.

What has been said about the possible memory bottleneck in RL machines applies equally to RC. The RC devices can make very efficient use of the available bandwidth by data reuse and a stream computing model, but the chip should have several IO channels connected to memory banks to provide sufficient base bandwidth for the massive parallelism. We recall that an ambitious solution to the memory gap problem is the integration of both the memory and the PEs closely coupled on the same die (Section 3.2.5 on page 96). Also the stream processors offer another way to reduce memory workload (Section 3.2.4 on page 93). Although RC devices, stream processors and PIM chips are all still targets of ongoing intensive research, the question arises how do they compare in terms of performance on different applications. In [Suh et al., 2003] we find a performance analysis of three representatives on memory intensive DSP kernels. Not surprisingly PIM performs well on particularly memory intensive kernels when the large internal memory bandwidth comes into play. The tested stream processor has an architectural preference towards more computationally intensive tasks where it gains the best score, but also memory use in the other tests could be optimized as compared to a microprocessor. The RC array of PEs, which consists of small processors with local cache in this case, performs best on average. These results stem from the high flexibility which allows to choose the appropriate programming and communication model for each application. The high flexibility of RC arrays has been also demonstrated by simulating a dedicated stream processor and a speculative multi-processor on the Smart Memory architecture [Mai et al., 2000], where the incurred performance loss could be mainly attributed to a greater number of local PEs or faster memory in the original architectures.

3.3.2 No Exponential is Forever

’No Exponential is Forever ... but We Can Delay ”Forever”’ was the accurate title of the opening keynote on the past, present and future of integrated circuits by Gordon Moore at the 50th International Solid State Circuits Conference 2003 [Moore, 2003]. Predictions in an exponentially developing environment are very difficult and many technological visions of the future have proven wrong. Accordingly, *Moore’s Law* had been announced dead several times already and Gordon Moore himself admits that some of his extrapolations into the future did not come true. But the general exponential process has continued, fed by an unending stream of new ideas which over and over again have overcome technological barriers standing in the way of the ongoing progress.

On the other hand, there is definitely an end to transistor miniaturization at the latest on the atom level. We may come to a stop well before that because of also exponentially growing fabrication costs and the power leakage problem. Currently, the train is still on track and the International Technology Roadmap for Semiconductors (ITRS) 2003 [SEMATECH, 2003] indicates that though the rate of the exponential growth may have slowed down to a 3 years transistor count doubling, the exponential growth will most likely continue to at least the end of this decade. Beyond that, major innovations in the design and fabrication of electronic integrated circuits will be required to uphold Moore's Law, and this innovations have not only to be technically feasible but also economic. The world wide initiatives for the Extreme Ultra-Violet (EUV) lithography [Gwyn and Silverman, 2003] paving the way for 40nm feature sizes and below, give an idea of the immense expenses associated with the intended adherence to the exponential growth.

Finally, around 2020 we will probably be so close to the fundamental limit of a charge-based architecture, that other state variable representations will have to be considered. In contrast to previous ITRS reports the variety of new, not charge-based concepts has increased, but this makes it even harder to look into their future and to judge which or whether any of them have the potential to delay the end of the exponential for some more years. To a small extent we have already touched future design concepts with the array of PEs and local interconnects in reconfigurable architectures, since in nano-scale devices (10^{12} elements/cm^2) a regular and homogenous array arrangement will probably be the only one accessible to economic fabrication. Insofar nano-scale devices would also favor image processing applications which exhibit local neighbor communication and a regular data-flow (Section 2.6 on page 63). However, other large scale, defect- and fault-tolerant programming models will be needed than the demanding but comparably simple top-down mappings used for the implementations on reconfigurable hardware in the next chapter.

3.4 Conclusions

The *memory gap* has become the main obstacle to fast data processing of large data amounts. Almost all innovations in hardware architectures seek to overcome or at least alleviate the effects of insufficient memory performance. The high latencies of the memory chips can be traced back to the memory core structure (Section 3.1.1.1), which is unlikely to change in the near future. Instead, techniques from the memory chip over the memory system to the processor design level have been developed, which help to hide the latencies. Ideally, one obtains a seamless stream of data after an initial latency, and thus a maximal sustained bandwidth (Section 3.1.1.2). But the latencies can be hidden only when the requested addresses are known long before the need for the data. So the algorithm must exert either a regular or a highly predictable data-flow. Even when latencies are hidden, the bandwidth of memory chips cannot keep pace with the throughput of the processing elements (PEs). Only at the system level one could achieve an exponential increase of bandwidth (Section 3.1.2), but this still does not satisfy the requirement of the processing elements (PEs), and thus a whole hierarchy of differently fast memories aimed at frequent data reuse emerged (Section 3.1.3).

The instruction-stream-based (ISB) computing paradigm does not attach sufficient importance to the mismatch of computing and memory performance. Large caches and multiple data processing improve the situation, but only scratch the potential of parallelization (Section 3.2.3). Distributed memory systems do a better job in this respect, as they achieve a higher level of parallelism in the spatial domain (Section 3.2.2). Architectures based on data-stream-based (DSB) processing go even further by exploiting parallelization in breadth and depth already at the finest level (Section 3.2.4). High numbers of independent PEs provide an enormous throughput amplified by flexible use of pipelining (Section 3.2.1). The direct coupling of PEs and memory cells in the Processor-in-Memory (PIM) designs seeks to overcome the bandwidth shortage in principle (Section 3.2.5).

Considering contemporary architectures, micro- and DSP-processors still hold a dominant position, but the superior performance of DSB architectures in data intensive applications gains increasingly more attention (Sections 3.3.1). But the acceptance of a technology depends strongly on the availability of comfortable tools to use it. Field Programmable Gate Arrays (FPGAs) have matured farthest in this respect. The demand for performance also drives the research on development tools for Reconfigurable Computing (RC), PIM and stream processors. The adaption of these new technologies is still slow, since the immense parallelism they offer, asks for different programming models and ultimately new High Level Languages (HLLs). But in the long term this is a small price to pay for their ability to fight the memory gap despite high scalability. They even hint at the future processing on nano-scale devices, where massive parallelism will be the predominant feature (Sections 3.3.2). So the architectures used for the image processing problems in the next chapter assign exclusively to the DSB model.

4 Hardware Efficient Implementations

Contents

Figures

Tables

This chapter presents efficient implementations of quantized PDE based image processing tasks on data-stream-based (DSB) hardware architectures. The implementations exert hardware efficiency by exploiting the performance characteristics and respecting the functional restrictions of the hardware platforms.

For three different platforms, the utilization for PDE solvers is described and efficient implementations are presented. The focus lies on image processing with graphics hardware, covering the whole range from simple standard OpenGL functionality back in 1999 to the high level shader programs from 2003, which eliminated many of the technical constraints present in previous years. The implementations in fine and coarse grain reconfigurable hardware reside even closer to the direct hardware programming level using either a Hardware Description Language (HDL) or a Native Mapping Language (NML) to configure the appropriate hardware state. This approach offers additional optimization opportunities which result in utmost performance of the algorithms, and the range of the applications is only restricted by the size of the available hardware devices. At the end we compare the pros and cons of the different implementations where not only performance but also the flexibility and programmability of the architectures play an important role.

4.1 Graphics Hardware

Graphics hardware has undergone a rapid development over the last 10 years. Starting as a primitive drawing device it is now a major computing resource. Together with the increasing functionality more and more general problems have been mapped to the graphics architecture, see [GPGPU] for an elaborate overview. We discuss the implementations of the PDE based image processing models from Chapter 2. But first we outline the technological development and logic layout of graphics hardware, and its usage for scientific computations. Parts of this exposition have been published in [Rumpf and Strzodka, 2005].

4.1.1 Technology

The rapid development of graphics hardware introduced a lot of features and key words. The following sections sketch the evolution. Omitting some details we outline the common structures and the key innovations of the different generations.

4.1.1.1 Development

Up to the early 1990s standard graphics cards were fairly unimpressing devices from a computational point of view, although having 16 colors in a 640x350 display (EGA) as opposed to 4 colors in a 320x200 display (CGA) did make a big difference. Initially the cards were only responsible for the display of a pixel array prepared by the CPU of the computer. The first available effects included the fast changing of color tables, which enabled color animations and the apparent blending of images. Then the cards started to understand 2D drawing commands and some offered additional features like video frame grabbing or multi-display support.

The revolutionary performance increase of graphics cards started in the mid 1990s with the availability of graphics accelerators for 3D geometry processing. The already well established game market, welcomed this additional processing power with open arms and soon no graphics card would sell without 3D acceleration features. Since then the GPU has taken over more and more computational tasks from the CPU. The performance of GPUs has grown much faster than that of micro-processors, doubling performance approximately every 9 months, which can be referred to as ***Moore's Law squared***.

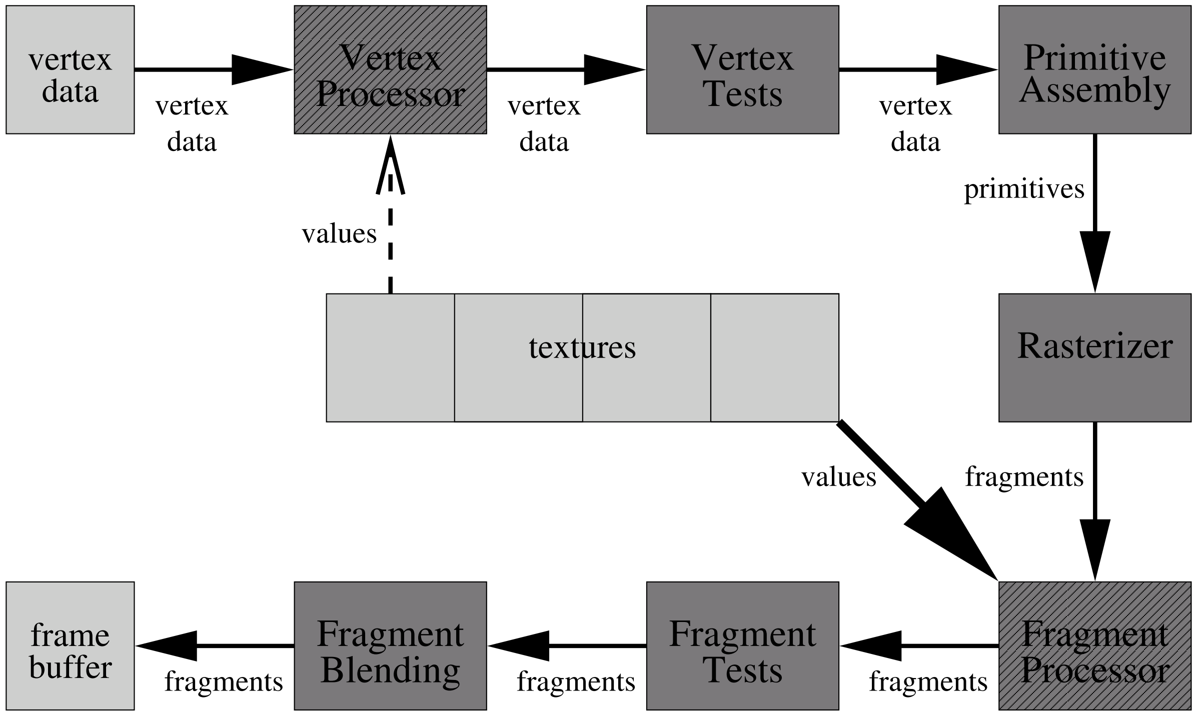
During the late 1990s the number of GPU manufacturers boiled down to very few, at least for PC graphics cards. Although other companies try to gain or regain ground in the market, NVIDIA and ATI have been clearly dominating it both in performance and market shares for several years now. In the following discussions we therefore primarily cite their products. Concerning the market, we should mention that actually Intel is the largest producer of graphics chips in form of integrated chip-sets. But these are inexpensive products and rank low on the performance scale. Consequently, we will deal only with GPUs on dedicated graphics cards.

Together with the reduction of GPU designers, the number of different Application Programming Interfaces (APIs) to access their functionality also decreased. The OpenGL API [Ope] and the DirectX API [Mic] are the survivors. The API guarantees that despite the different hardware internals of GPUs from different companies, the programmer can access a common set of operations through the same software interface, namely the API. The proprietary graphics driver is responsible for translating the API calls into the proprietary commands understood by the specific GPU. In this respect the API is similar to an operating system which also abstracts the underlying hardware for the programmer and offers standardized access to its functionality, although an operating system does more than that.

If the hardware offers new features and is downward compatible, the old API still functions, but it lacks the new functionality. The use of the new features in the new API, however, results in an incompatibility with older hardware. Therefore, programmers are reluctant to use new features as long as they expect a significant demand for their applications on older hardware. The hardware vendor can promote the use of the new API by emulating the new hardware features in software on older systems. But this may turn out very demanding or impractical if the software emulation is too slow. So in practice programmers stick voluntarily to very low requirements for the hardware and do not bother about incompatibility issues. Only the time critical parts of the code are sometimes implemented for each hardware standard separately and chosen dynamically upon hardware identification. The above applies both to programs for different versions of an operating system and programs (mainly games) for different versions of graphics APIs. However, graphics hardware has evolved much quicker and game performance is often a critical factor, such that the changes of API versions and the lowest common requirements are moving faster than in the micro-processor market.

OpenGL and DirectX have been incorporating the quickly evolving feature set of GPUs differently. OpenGL uses a very flexible extension system. Each vendor can expose the whole functionality of its hardware product by proprietary extensions to the API. The OpenGL Architectural Review Board (ARB) [Ope] which includes the main players in the graphics field helps in the standardization of these extensions to prevent the undermining of the common interface idea through too many incompatible proprietary extensions. In practice, first the proprietary extensions appear and then the standard access points evolve over time. The different versions of DirectX on the other hand, are prescribed by Microsoft and thus simply define a fixed set of requirements. Naturally, these requirements are discussed with the GPU designers beforehand. If the hardware supercedes them quantitatively, then DirectX often allows the use of these additional resources, but qualitatively new features have to wait for the next API generation. So the DirectX API changes more or less step in step with the new graphics hardware generations, while OpenGL evolves continuously, first on proprietary and subsequently on ARB paths. Currently (2004), OpenGL also undergoes the first major revision since 1992 from the 1.x versions to version 2.0 [Ope] in an attempt to include many of the already well established extensions into the core and prepare the API for future developments.

Figure 4.1 A simplified diagram of the graphics pipeline. Light gray represents data containers, dark gray processing units. The emphasized Vertex Processor (VP) and Fragment Processor (FP) are the units which evolved most in the graphics pipeline over the years, up to the stage where they accept freely programmable shader programs as configurations.
Since the widespread support for one pass multi-texturing (1998), the thick arrow from the textures to the FP represents the largest data streams in the pipeline. Accordingly, the FP consumes the majority of resources in a GPU. The access to textures from the VP is a recent feature (2004) as well as the upcoming full interchangeability of the data containers in the pipeline, i.e. a 2D data array can serve as an array of vertex data, a texture, or a destination buffer within the frame-buffer.



4.1.1.2 Graphics Pipeline

The ***Graphics Processor Unit (GPU)***, the central computational chip on a graphics card, may be seen as a restricted form of a stream processor (cf. Section 3.2.4 on page 93). With a number of commands one configures a state of the graphics pipeline in the GPU and then sends data streams through that pipeline. The output stream is visualized on the screen or sent again through the pipeline after a possible reconfiguration. Although graphics cards have not been seen in this context from the beginning, the current developments show a clear tendency towards a general stream processor. Therefore, we want to uphold this view when discussing the different GPU generations in the next section and subsequently.

A schematic view of the graphics pipeline is presented in Figure 4.1. The abstraction omits some details but encompasses the whole evolution of the pipeline in a common diagram. The changes from one graphics hardware generation to another can be identified by describing the increased flexibility and functionality of the components in the pipeline. For the sake of a consistent presentation, we use the current terminology for the components, even if it has not been used for the older pipeline generation back then. First let us describe the operational task of the individual components:

- vertex data

  We need an array which defines the geometry of the objects to be rendered. Beside the vertex coordinates the vertex data may also contain color, normal and texture coordinate information (and a few more parameters). Although the data may be specified with 1 to 4 components, both coordinates (XYZW) and colors (RGBA) are internally always processed as 4-vectors. During the graphics hardware evolution mainly the choices for the places where the vertex data can be stored (cacheable, AGP or *video memory*) and the efficiency of handling that data increased. The modern Vertex Buffer Objects (VBOs) allow to specify the intended use and let the graphics driver decide which type of memory is ideally suited for the given purpose.

- ***Vertex Processor (VP)***

  The VP manipulates the data associated with each vertex individually. Over the years the number of possible operations increased dramatically. In the beginning only multiplications with predefined matrices could be performed. Nowadays, the VP runs assembly programs on the vertex data and the new generation (May 2004) has already a restricted texture access from the VP. But each vertex is still processed individually without any implicit knowledge about the preceeding or succeeding vertices, although this may also change in the future.

- vertex tests

  Vertex tests determine the further processing of geometric primitives on the vertex level. They include mainly the back-face culling, which eliminates polygons facing backwards (if the object is opaque one cannot see its back) and clipping, which determines the visible 3D space with an intersection of several 3D half spaces, defined by clipping planes. The vertex tests are still controlled by parameters and experienced only quantitative improvements in the number of clipping planes over time.

- primitive assembly, rasterizer

  The geometric primitives which can be rendered are points, line segments, triangles, quads and polygons. Each vertex is processed individually and the clipping of primitives may introduce new vertices such that primitives have to be reassembled before rasterization. Also, for simplicity the rasterizer operates in many graphics architectures exclusively on triangles, so other primitives must be converted into a set of triangles before processing. Given a triangle and the vertex data associated with each of its vertices, the rasterizer interpolates the data for all the pixels inside the triangle. The resulting data associated with a pixel position is called a ***fragment***. From the beginning the rasterization could be controlled with parameters, for example defining patterns for lines and the interior.

- ***textures***

  Textures are user defined 1D to 4D (typically 2D) data arrangements stored in video memory of the graphics card. Their elements which can have up to four components (RGBA) are called ***texels***. In general the dimensions of all textures must be powers of 2, with the exception of rectangular textures in 2D since 2001. Only recently (2004) a general extension for non-power of 2 textures has been introduced.

  Input images of a problem are usually represented as textures on the graphics card and their values are processed by the FP and fragment blending. Over the years quantitative improvements of textures included their maximal number, their maximal size and the precision of the used *fixed point number* format. Qualitative improvements are the support of various dimensionalities, the different access modes, the *floating point number* format, and the flexibility in the creation and reuse of texture data in different contexts. From the modern point of view textures represent just a special use of data arrays which can serve as input to the FP (texture mode), as the destination for the output stream of the graphics pipeline (output mode), or even as an array defining vertex data (vertex mode).

- ***Fragment Processor (FP)***

  The FP manipulates the individual fragments. Similar to the vertices, each fragment is processed independently of the others in the same data stream. With the interpolated texture coordinates, the FP can access additional data from textures. The functionality of the FP has improved enormously over the years. In a qualitative sense the range of available access modes of texture data and operations on these values in the FP has grown rapidly culminating in a FP controlled by assembly or high level code with access to arbitrary texture positions and a rich set of mathematical and control operations. In a quantitative sense the number of accessible textures and the number of admissible fragment operations has increased significantly.

- ***frame-buffer***

  The frame-buffer is the 2D destination of the output data stream. It contains different buffers of the same dimensions for the color, depth and stencil (and accumulation) values. Not all buffers need to be present at once. Also, each buffer allows certain data formats but some combinations may not be available. There exists at least one color buffer, but typically we have a front buffer, which contains the scene displayed on the screen and a back buffer where the scene is built up. Over the years mainly the maximal size, the number and the precision of the buffers increased. The recent development (2004), already sketched in the texture item, regards the frame-buffer as an abstract frame for a collection of equally sized 2D data arrays. After the rendering the same 2D data arrays may be used as textures or vertex data.

- fragment tests

  Equivalent to the vertex tests for vertices, the fragment tests determine whether the current fragment should be processed further or discarded. But the fragment tests are more numerous, powerful and some of them allow a comparison against the values stored at the associated pixel position of the fragment in the depth or stencil buffer, and also a restricted manipulation of these values depending on the outcome of the tests. This direct manipulation is a particular benefit, as the FP does not offer it.

- ***fragment blending***

  Before the FP became a powerful computational resource, computations were mainly performed by different blending modes. The blending operation combines the color value of the fragment with the color value in the color-buffer controlled by weighting factors and the blending mode, e.g. convex combination of the values with a certain weight. Blending became less popular in the last years because on most GPUs it did not support the higher precision number formats as the much more powerful FP did. But currently the higher precision blending support is increasing again. The advantage of blending is the direct access to the destination value in the frame-buffer, which is officially not supported for the FP.

  The blending modes are continuous functions of the input values, additionally logical operations can be performed at the end of the pipeline, but these are very seldomly used, as they have received no hardware support from the GPUs builders.

Figure 4.1 on page 112 visualizes the streaming nature of GPUs and the *parallelism in depth* innate to the *pipelining* concept . Because of the independent processing of data elements in a data stream, GPUs also exercise *parallelism in breadth* (Section 3.2.2 on page 87). The dual programming model of data-stream-based (DSB) architectures (Table 3.2 on page 85) applies. The *configware* consists of a large number of parameters which define the behavior of the various stages of the pipeline, and newly also assembly or even High Level Language (HLL) programs which configure the VP and FP. The *flowware* is given by function calls to the graphics API, which are embedded in a software program for the CPU of the PC in which the graphics card resides.

For the efficiency of this approach it is crucial that the same configuration is applied to large data streams. Then the addresses of the required data from the textures form a regular pattern and by using catenated *burst mode* reads a seamless stream of data hiding all intermediate memory access latencies (Table 3.1 on page 74 without tCL) can be obtained (Figure 3.3 on page 76). Graphics cards also quickly adopted the *bandwidth* doubling Double Data Rate (DDR) memory and the efficient *bank interleaving* access for maximal *sustained bandwidth* (Section 3.1.1.2 on page 76). For the same purpose they use memory chips of high (currently 32 bit) *memory depth* (Section 3.1.1 on page 71) which line up to very wide (currently 256 bit) data buses (Section 3.1.2 on page 79). But for the overall performance not only the bandwidth of the memory but also the ***throughput*** of the processing elements (PEs) is decisive (Section 3.2.1 on page 84). In contrast to cache equipped instruction-stream-based (ISB) architectures, GPUs used to have an almost equally balanced bandwidth and throughput, which is most suitable for problems with low ***computational intensity***. The latest (2004) hardware generation has an overbalance on the computational side, because the higher programmability of the pipeline is increasingly used to solve more computationally intensive tasks.

For regular access patterns to the textures the efficiency and synchronization of the graphics pipeline is so high, that the achievable performance for a simple pipeline configuration almost matches the theoretic *peak bandwidth* of the memory system. Since 2001 dependent and offset texture reads destroy the static nature of texture reads. But despite dynamically computed addresses for texture access the efficiency of DSB paradigm does not break down. Especially if the computed addresses as a whole represent a smooth deformation of the texture source, e.g. translation, shear, magnification, then small caches suffice to hide the additional latencies.

4.1.1.3 Classification

Because of the almost synchronous evolution of the DirectX (DX) API and the generations of graphics hardware in recent years, it is easiest to classify GPUs according to the highest DX version which they support. Actually we are only concerned with the Direct3D API, but Microsoft releases the different APIs in a bundle, so one usually refers to the version of the whole release. From DX8 on, one can further differentiate by the functionality of the Vertex Shaders (VSs), which configure the VP, and the Pixel Shaders (PSs), which configure the FP. This DX,VS,PS classification is very common and we will refer to it, although all our implementations use the OpenGL API. Therefore, the new features of each hardware generation are accompanied by the corresponding OpenGL names for the extentions. This has also the advantage of a precise identification of certain extensions in specific GPUs, whereas the fractional numbers for VSs and PSs give only a hint at the increased functionality. The evolution of the functionality of the FP is also outlined in mathematical terms in Section 4.1.2.3 on page 124.

We have tried to list the main GPU lines in each generation by studying the technical specifications and various benchmark results. The most important factor for scientific computing on GPUs is usually the raw fragment performance. Benchmarks, however, are strongly influenced by the choice of tested functionality (many testers use games as benchmarks), the quality of the available API implementations at this time and the properties of the test computer system. As a consequence we excluded tile based renderers, notably the Kyro series from PowerVR Technologies, which perform very well in typical games as compared to their raw processing power, but this raw power is decisive for scientific computing. Also, it is quite unfair to directly compare the products released at the beginning and end of a year, since one year has meant a lot in the development of graphics hardware. So the dates and examples GPUs give a good orientation, but are not suitable for a definitive ranking of the GPUs. The overview is partially based on [Fernando and Kilgard, 2003].

- Proprietary graphics systems,

  e.g. Evans & Sutherland (E&S) [Evans & Sutherland, 2004], Silicon Graphics Inc. (SGI) [Lenerz, 2004]

  As soon as computers were used for simulation or analysis of data sets, there was a need to visualize the results. Without a standard at hand, application specific solutions dominated in the early days of graphics. Nowadays, standard GPUs have become very powerful and easily outperform specialized graphics workstations from years ago, but the segments of professional graphics accelerators and proprietary graphics systems still exist. Some professional cards now use (almost) the same GPUs as their gaming equivalents, but their focus goes much further in the direction of reliable and qualitative imaging, which concentrates more work on the graphics driver development than the pure hardware performance. Proprietary graphics systems are neither obsolete, as the standard GPUs are not designed for collaborative work on huge data sets. But even when standard hardware components are used for large scale visualization, similar to the parallel computing, the proper distribution and processing of data is by no means trivial.

- First accelerators for OpenGL, 1992-1997,

  e.g. AccelGraphics AG300, DEC workstations with ZLX graphics, IBM Freedom Series /6000, Intergraph TD Series with GLZ graphics, HP Freedom Series, SGI Indy/Indigo/Onyx, 3DLabs GLINT 300SX.

  In 1992 SGI introduced the OpenGL API. This turned out to be a significant step in the evolution of graphics software and hardware. Within a few years most major players in the graphics field, often with the support of E&S, offered OpenGL implementations for their proprietary graphics accelerators. In this way graphics applications became system and platform independent and the success of OpenGL up to now highlights the importance of this change. OpenGL 1.0 already defined the graphics pipeline and its basic functionality as described in the previous section, although many features were not supported in hardware at first.

- First accelerators for DirectX, 1996-1997,

  e.g. 3dfx Voodoo, 3DLabs Permedia2, NVIDIA Riva 128, ATI Rage Pro.

  In 1995 Microsoft released the first version of DirectX and from 1996 on it contained the access points to 3D acceleration. Although there were quite a few games around at this time already, the first 3D accelerator Voodoo by 3dfx boosted both the game and graphics hardware market. All companies were eager to promote their GPUs with the 3D label even if little actual 3D acceleration was present. Different APIs were used at this time (e.g. OpenGL, DirectX, Glide, Metal, MGA) and support for the increasingly popular DirectX was often realized by a translation layer. Naturally, the first DirectX versions lagged behind the already matured OpenGL API. DX5 shipped with Window98 and after another major overhaul in DX6 the development started to synchronize well with the graphics hardware generations.

- DX6 GPUs, 1998-1999,

  e.g. 3dfx Voodoo2/Voodoo3, Matrox Millenium G400 /MAX, NVIDIA Riva TNT/TNT2, ATI Rage Fury 128/MAXX.

  These GPUs offered multi-texturing (dual) in one pass (`GL_ARB_multitexture`) and the RGBA8 format for textures and frame-buffers resolving each of the four color channels in 8 bit had been adopted. At this time also hardware support for the stencil and depth buffers and different blending modes (`GL_EXT_blend_color, GL_EXT_blend_minmax, GL_EXT_blend_subtract`) increased.

- DX7 GPUs, 1999-2000,

  e.g. 3dfx VSA-100 Voodoo4/Voodoo5, S3 Savage 2000 /+, NVIDIA GeForce256/GeForce2, ATI Radeon.

  Transform&Lighting was the key hardware feature of this period, which basically meant the introduction of a parameter controlled VP. From the computational point of view, however, the increased functionality of the FP was more important (`GL_EXT_texture_env_combine`). It allowed more arithmetic operations in the texture environments and was further enhanced by the prominent feature of bump mapping (`GL_EXT_texture_env_dot3`). Conditional assignments in the FP appeared (`GL_NV_register_combiners`). Support for 3D textures (`GL_EXT_texture3D`) and environment mapping (`GL_EXT_texture_cube_map`) also evolved.

- DX8 (VS1, PS1) GPUs, 2001-2002,

  e.g. 3DLabs Wildcat VP, Matrox Parhelia 512 (VS2, PS1), NVIDIA GeForce 3/4, ATI Radeon 8500.

  These GPUs introduced the assembly programs for the VP (`GL_EXT_vertex_shader, GL_NV_vertex_program, GL_ARB_vertex_program`) and highly restricted programmability for the FP (`GL_ATI_fragment_shader, GL_NV_texture_shader, GL_NV_register_combiners`). Certain textures allowed non-power of 2 dimensions (`GL_NV_texture_rectangle`). System dependent extensions enabled rendering to *pbuffers* which can be bound as textures (`WGL_ARB_pbuffer, WGL_ARB_render_texture`).

- DX9 (VS2, PS2) GPUs, 2002-2004,

  e.g. S3 DeltaChrome S8, XGI Volari Duo V8, NVIDIA GeForceFX 5800/5900, ATI Radeon 9700/9800.

  The programmability of the VP (`GL_ARB_vertex_shader, GL_NV_vertex_program2`) gained function calls, dynamic branching and looping. But the PS2 model for the FP was an even larger leap forward, now allowing freely programmable assembly code operating on up to 16 textures (`GL_ARB_fragment_program, GL_NV_fragment_program`). High level languages (Cg [NVIDIA, 2002], DX9 HLSL [Mic, 2003], GLSL [Ope, 2004]) allow easier programming of the VP and FP. Floating point formats for textures and pbuffers appeared (`GL_ATI_texture_float, GL_NV_float_buffer`). Vertex Buffer Objects (VBOs) allow the efficient transfer and reuse of vertex data (`GL_ARB_vertex_buffer_object`). Rendering to several pbuffers is possible (`GL_ATI_draw_buffers`).

- DX9+ (VS2-VS3, PS2-PS3) GPUs, 2004,

  e.g. 3DLabs Wildcat Realizm (VS2, PS3), NVIDIA GeForce 6800 (VS3, PS3), ATI Radeon X800 (VS2, PS2).

  In the VS3 model, the VP gains additional functionality in form of a restricted texture access and more functionality for register indexing (`GL_NV_vertex_program3`). The PS3 FP now also supports the features of function calls and restricted forms of dynamic branching, looping and variable indexing of texture coordinates (`GL_NV_fragment_-`

program2). Pixel Buffer Objects (PBOs) allow the use of textures as vertex data (GL_EXT_pixel_buffer_object).

- WGF 1.0, 2006?
  The next Windows generation (Longhorn [Microsoft]) will contain a radically new graphics interface labeled Windows Graphics Foundation (WGF). The main expected features are a unified shader model, resource virtualization, better handling of state changes and a general IO model for data streams. Future GPU generations will probably support all these features in hardware.

Our implementations cover the different functionality levels from DX6 to DX9. The used graphics hardware includes SGI Onyx2, NVIDIA GeForce 2/3/FX5800 and ATI Radeon 9800.

4.1.2 Computations

After getting to know the development and internal structure of GPUs we must describe how the actual processing takes place. This includes the description of the general data-flow on the card, the available number formats for data representation and the evolution of the PEs in the pipeline.

4.1.2.1 Data-Flow

The general data-flow in a GPU is prescribed by the graphics pipeline (Figure 4.1 on page 112). The standard data path from the main memory and the textures to the *frame-buffer* has been fast since the beginning. But in iterative PDE solvers we need more than one pass and intermediate results must be reused for subsequent computations. This means that the content of the frame-buffer must be resent through the graphics pipeline again and again. The efficiency of this non-standard data-flow has improved only slowly and a fully satisfactory implementation is still in development. There are four possibilities to further process the results from the frame-buffer:

- Read-back (glReadPixels).
  We can read the selected content of the frame-buffer back to the main memory. This is a slow operation, because data transfer has always been optimized in the direction from main memory to the graphics card. With the PCI Express (PCIe) bus with a symmetric bandwidth in both direction this has finally change in 2004. But even then the available bandwidth on-card is higher than over the bus, so transferring data to the main memory and back onto the card is inefficient. Data should be read-back only if it requires analysis by the CPU.

- Copy-to-texture (glCopyTexSubImage1D/2D/3D).
  The frame-buffer content can be used to define parts of a texture. This also requires copying of data but the high data bandwidth on-card makes this operation much faster

than the read-back. Unfortunately, the graphics drivers did not support the fast texture redefinition well until late 2000.

- Copy-to-frame-buffer (`glCopyPixels`).
  It is possible to copy data from the frame-buffer onto itself. Because the copied data enters into the pipeline at the height of the FP and many of the per-fragment operations can be applied to the data stream, it is actually not raw copying but re-rendering of the data. Not all functionality of the FP can be used, however, since texel operations associated with the texture environment (`glTexEnv`) are not executed. Therefore, also the new programmable FP programs cannot applied to the data. Copying can be performed between different color buffers and later (2001) partial support for the depth buffer appeared. Similar to the copy-to-texture, at this time also the Copy-to-frame-buffer greatly improved in terms of speed in the general case, when the fragment pipeline is configured with different operations. Before only the raw copying was fast.

  Considering the necessary data movements this functionality is superior to the previous ones, as the re-rendering directly applies the new operations to the data instead of copying it to some other place first. The following option also avoids the copying, but allows to use the entire pipeline for the re-processing.

- Render-to-texture (`WGL_ARB_pbuffer, WGL_ARB_render_texture`).
  One can allocate a ***pbuffer***, i.e. an additional non-visible rendering buffer like the frame-buffer, which serves as the destination for the output data stream. As soon as the pbuffer is not a render target any more, it can be used as a texture, which ultimately means that one renders directly to a texture. The only problem with pbuffers is that they carry a lot of static information which causes a performance penalty for every switch between their use as a data source (texture) or data destination (frame-buffer).

- ARB superbuffers.
  Current graphics driver development addresses the problem of slow pbuffer switches by introducing a new, light-weight mechanism for using raw data arrays as source or destination at various points in the graphics pipeline. The idea is to define a memory array together with some properties which describe the intended usage. The graphics driver will then decide where to allocate the memory (cacheable, AGP or *video memory*) depending on these properties. To some extent the functionality is already available with the Vertex Buffer Object (VBO) and Pixel Buffer Object (PBO) extensions, but the OpenGL ARB superbuffer group works on a more general and fully flexible solution.

For all early implementations the access to the frame-buffer was a major bottleneck, as the performance factor between drawing a texture into the frame-buffer and reading it back to the texture was more than 50. Our early implementations use the copy-to-frame-buffer, later the copy-to-texture, and the newer ones the render-to-texture mechanism. If few pixel values have to be retrieved the read-back is used, but computations on the entire images are always performed on the card, so that images never have to be read back to main memory.

4.1.2.2 Number Formats

For a long time the R5G6B5 format requiring 16 bit for the colors red, green and blue has been used for gaming. Even today this can be still done, if the game is bandwidth bound. But the processing elements (PEs) in the available GPUs do not gain any performance from the low precision format anymore. Already the first DX6 GPUs offered the RGBA8 unsigned fixed point format in 1998, and this has remained a standard till nowadays. Soon after, a sign has been added for the internal computations, but negative results could not be stored as such. Little has changed until 2001, when DX8 GPUs introduced some proprietary higher precision formats, in particular several signed ones, but these were not supported throughout the pipeline and thus of little help in iterative computations. Over a year later the DX9 GPUs offered floating point precision throughout the pipeline and added new fixed point formats again. Graphics workstations, e.g. SGI Onyx, supported 12 and 16 bit unsigned fixed point formats since the mid 1990s. The result of these developments is a confusing number of incompatible fixed point formats nowadays (Table 4.1 on the following page).

Nearly all introduced fixed point formats represent numbers in $[-1, 1]$, and initially the unsigned formats dominated, covering the range $[0, 1]$. But they all, even those with 16 bit precision, suffer from insufficiencies in the exact representation of certain values:

- The values represented by the RGBA8 format.
  The RGBA8 format is still the most widely used format, as it offers a good compromise between precision and memory consumption. Inaccurate conversions must be performed to represent its values in most of the new formats, including the floating point formats.
- The values -1,0,1.
  These are the neutral elements of addition, multiplication and its negative. Failure to represent them exactly, makes it impossible to preserve the values of certain regions in iterative computations without the introduction of performance deteriorating exceptions.
- The values $\pm 2^{-1}, \pm 2^{-2}, \dots, \pm 2^{-7}$.
  Most number formats based on the binary system can only represent fractions with a power of 2 denominator exactly. This may not be much, but it is a very desirable property to be able to invert the power of 2 multiplications exactly. This applies especially to $\frac{1}{2}$ and $\frac{1}{4}$ which appear naturally in many formulas.

The apparent incompatibility of these conditions is a general problem of fixed point number systems and has to do with the choice of the *machine epsilon* (see Section 2.2.2.2 on page 30). It can be solved by choosing a machine epsilon whose denominator contains the appropriate factors. A ***virtual signed 16 bit format*** which fulfills the above conditions and serves as a superset of all 8-9 bit formats has been introduced in [Strzodka, 2002]. No native hardware implementation of this format exists. In this paper an unsigned RGBA8 format is used to represent the two virtual signed 16 bit numbers. With sufficient processing capabilities of the FP (DX8 GPUs) the usual arithmetic operations of addition, subtraction, multiplication and arbitrary function evaluation can be performed in one rendering pass. In DX9 floating point

Table 4.1 Comparison of fixed point number formats in graphics hardware. The formats are grouped in pairs with a subset relation within each pair. The main point of the table is to show that the numbers represented by one pair of formats in general cannot be represented by the other pairs. Therefore we have so many '-' signs below.
The unsigned 8 bit format is still the most common choice among the fixed point formats. Some machines offer also a 12 bit format build up correspondingly to the other unsigned formats. The signed 9 bit format is only an internal format used for computations, but it is the only one with an exact representation of -1,0,1, although it fails on 0.5 and other negative powers of 2. It was superceded by an internal 12 bit format which worked in the same way but covered the interval $[-8,8]$. Into the middle of the last pair one could also amend another internal signed 12 bit format with the formula $a/2^{10}$ and the codomain $[-2,2-\frac{1}{1024}]$. It suffers from the same problems as the other two, i.e. no representation of the upper bound integer and not a superset of the standard unsigned 8 bit format.

	unsigned 8 bit	unsigned 16 bit	expanded 8 bit	signed 9 bit	signed 8 bit	signed 16 bit
formula	$\frac{a}{2^8-1}$	$\frac{a}{2^{16}-1}$	$\frac{2a-1}{2^8-1}$	$\frac{a}{2^8-1}$	$\frac{a}{2^7}$	$\frac{a}{2^{15}}$
range of a	[0,255]	[0,65535]	[0,255]	$[-256{,}255]$	$[-128{,}127]$	$[-32768{,}32767]$
codomain	[0,1]	[0,1]	$[-1{,}1]$	$[-1-\frac{1}{255},1]$	$[-1,1-\frac{1}{128}]$	$[-1,1-\frac{1}{32768}]$
machine epsilon $\varepsilon^{\mathcal{Q}}$	$3.9\cdot10^{-4}$	$1.5\cdot10^{-5}$	$7.8\cdot10^{-4}$	$3.9\cdot10^{-4}$	$7.8\cdot10^{-4}$	$3.1\cdot10^{-5}$
represented number	Value of a or non-representability ('–').					
$\mathbf{+1.0}$	$\mathbf{255}$	$\mathbf{255\cdot257}$	$\mathbf{255}$	$\mathbf{255}$	**–**	**–**
$+0.99998..$	–	$255{\cdot}257{-}1$	–	–	–	–
$+0.99996..$	–	–	–	–	–	$128\cdot256-1$
$+0.99607..$	254	$254\cdot257$	–	254	–	–
$+0.99218..$	–	–	–	–	127	$127\cdot256$
$+0.99215..$	253	$253\cdot257$	253	253	–	–
$+0.5$	–	–	–	–	64	$64\cdot256$
$\mathbf{0.0}$	**0**	**0**	**–**	**0**	**0**	**0**
-0.5	–	–	–	–	-64	$-64\cdot256$
$-0.99215..$	–	–	-253	-253	–	–
$-0.99218..$	–	–	–	–	-127	$-127\cdot256$
$-0.99607..$	–	–	–	-254	–	–
$-0.99996..$	–	–	–	–	–	$-128{\cdot}256{+}1$
$-0.99998..$	–	–	–	–	–	–
$\mathbf{-1.0}$	**–**	**–**	$\mathbf{-255}$	$\mathbf{-255}$	$\mathbf{-128}$	$\mathbf{-128\cdot256}$

Table 4.2 Operations on encoded values for an affine mapping of number ranges. Because for a long time only the $[0,1]$ codomain was supported by the graphics number formats, one had to use an affine mapping onto this range to represent a larger interval $[-\rho,\rho]$. The table lists the operations which must be performed on the encoded values to obtain the correct results.

operations	operations on encoded values
$\longrightarrow$	$r : x \to \frac{1}{2\rho}(x+\rho) \quad \longrightarrow$
$a, b \in [-\rho,\rho]$	$r(a), r(b) \in [0,1]$
$a+b$	$r(a)+r(b)-\frac{1}{2}$
ab	$\frac{1+\rho}{2} - \rho\,(r(a)(1-r(b)) + r(b)(1-r(a)))$
$\alpha a + \beta$	$\alpha r(a) + (\frac{\beta}{2\rho} + \frac{1-\alpha}{2})$
$\max(a,b)$	$\max(r(a), r(b))$
$f(a_0,\dots,a_n)$	$(r \circ f \circ r^{-1})(r(a_0),\dots,r(a_n))$
$\sum_\alpha \alpha a_\alpha$	$\sum_\alpha \alpha r(a_\alpha) + \frac{1}{2}(1-\sum_\alpha \alpha)$
$\longleftarrow$	$r^{-1} : \rho(2y-1) \leftarrow y \quad \longleftarrow$

formats with at least 16 bit *mantissa* were introduced, so on DX8 hardware and beyond we can obtain at least 16 bit precision, but before that more care had to be taken.

The error analysis of the quantized solvers in Chapter 2 on page 13 uses the general assumption that the quantized number system $\mathcal{Q}$ contains the zero and that its positive numbers can be negated in the system (Section 2.2.3.1 on page 36). The introduced notation (Eq. 2.20 on page 31) covers only fixed point number systems which fulfill these conditions. In Table 4.1 on the preceding page only the relatively new signed formats comply with this. The signed 9 bit format was already made available in 1999, but it is only an internal format which did not fit into the 8 bit precision available at this time and it was supported only in the `GL_NV_register_combiners` extension. The other signed formats were also first proprietary and not supported throughout the pipeline, because the frame-buffer had still to use an unsigned 8 bit format.

So the question is how to provide a signed format including a zero encoded in the unsigned 8 bit format. If it is possible to separate the positive and negative numbers then one can use two textures and thus implicitly obtain a signed 9 bit format. Whenever the negative components are involved in the computations, a subtraction instead of an addition must be performed, but subtractive blending (`GL_EXT_blend_subtract`) has been supported since the DX6 GPUs. Another option is to implicitly shift the unsigned 8 bit format by $\frac{1}{2}$ (this cannot be represented exactly but corresponds to 128) and thus obtain the signed 8 bit format, but with an awkward scaling factor $\frac{255}{128}$. The scaling is irrelevant for addition and for multiplications with constants, but one must apply correcting factors when multiplying two general values encoded in this way. The appropriate shifts could be performed in the fragment pipeline with the pixel

Table 4.3 Setup and precision of floating point formats supported in graphics hardware. These formats were introduced with DX9, which required the graphics hardware to have a format with at least the FP24 precision. The *unit roundoff*, i.e. the upper bound on the relative error in approximating a real number, is $\frac{1}{2}\varepsilon^{\mathcal{Q}}$ (Eq. 2.18 on page 29).

	FP16	FP24	FP32
setup: sign, mantissa, exponent	s10e5	s16e7	s23e8
machine epsilon $\varepsilon^{\mathcal{Q}}$	$9.8 \cdot 10^{-4}$	$1.5 \cdot 10^{-5}$	$1.2 \cdot 10^{-7}$

transfer modes (`glPixelTransfer`) enabled during a copy-to-frame-buffer operation, the extended texture environment (`GL_EXT_texture_env_combine`) or the register combiners (`GL_NV_register_combiners`), whatever was available and supported in hardware. Table 4.2 on the preceding page lists the necessary operations in the general case when we encode the range $[-\rho, \rho]$ in $[0, 1]$ by an affine transformation.

Three different floating point formats have been introduced with the DX9 GPUs (Table 4.3). The DX9+ chips have PEs working with a subset of the standard IEEE s23e8 format (without denormalized numbers) throughout the pipeline, such that the lower precision format are only advisable if memory bandwidth or availability is critical. Image processing application usually do not operate on very different number scales, so that higher precision fixed point formats are sufficient if one designs the numerical schemes appropriately. However, in other scientific computing problems, especially when using adaptive grids which resolve certain areas with high spatial accuracy, the number format must also be adaptive. So the introduction of floating point textures and pbuffers was a huge step forward for GPUs in the direction of a general stream processor. Equipped with higher precision, many of the meticulous reformulations of numerical schemes for low precision fixed point numbers can be omitted. However, we should not forget that the handling of numbers on very different scales can introduce unexpectedly large roundoff errors when used without care (Section 2.2.2.1 on page 27).

4.1.2.3 Operations

In the preceeding two sections we have listed the different options for data-flow management and number representations. Now we turn our attention to the available operations depending on the GPU generation. Again, we will refer to the OpenGL extensions, but use the common DX classification from Section 4.1.1.3 on page 116.

Because the presentation concentrates on PC graphics cards we should mention that many of the innovations have been anticipated by vendors of graphics workstations, most notably SGI. Our first implementations of general purpose computations on graphics hardware in 1999 were based on the InfiniteReality2 graphics engine of the SGI Onyx2. It supported many different blending and texture environment modes, 3D textures, multi-texturing, 12 bit number formats, and other special features like convolution filters, color table, color matrix, or histogram extraction, although not all features delivered the appropriate performance or full functionality. Only around 2000 the rapid development in the PC GPUs market became so dominant, that these GPUs had to be seen as the innovative motor of the quickly evolving new functionality. However, for some time many of the new extensions were still adapted or extended versions of previously supported features by graphics workstations.

As we use a uniform grid for the data representation (Figure 2.1 on page 27) we can easily establish a one-to-one correspondence between the node values and the texel values in a texture. For this data representation we use the vector notation $\bar{V}_\alpha$ with a 2-dimensional multi-index $\alpha = (\alpha_x, \alpha_y) \in (0, \ldots, N_x - 1) \times (0, \ldots, N_y - 1)$ (cf. Section 2.2.1.2 on page 26). As the texel values are usually 4-vectors themselves, where appropriate we will indicate the components by $\bar{V}_\alpha.\Xi$, where Ξ can be any subset of $\{x, y, z, w\}$, e.g. $\bar{V}_\alpha.xyzw = vV_\alpha$ is the full 4-vector, $\bar{V}_\alpha.y$ is the y-component of $\bar{V}_\alpha$ and $\bar{V}_\alpha.xyz$ is a three component sub-vector of $\bar{V}_\alpha$.

In the following list we refer only to the fragment pipeline functionality because this is the part of the graphics pipeline with which we perform the image processing. The list is accumulative, i.e. GPUs with a higher DX version also support the features of previous DX generations.

- DX6 GPUs, 1998-1999.

$\bar{V}_\alpha \cdot \bar{W}_\alpha$	(`glTexEnv` or `glBlendFunc`)
lerp$(\bar{V}_\alpha, \bar{W}_\alpha, a) := (1-a)\bar{V}_\alpha + a\bar{W}_\alpha$	(`glTexEnv` or `glBlendFunc`)
$\bar{V}_\alpha \pm \bar{W}_\alpha$	(`glBlendEquation`)
min$(\bar{V}_\alpha, \bar{W}_\alpha)$, max$(\bar{V}_\alpha, \bar{W}_\alpha)$	(`glBlendEquation`)
$\big(f_x(\bar{V}_\alpha.x), f_y(\bar{V}_\alpha.y), f_z(\bar{V}_\alpha.z), f_w(\bar{V}_\alpha.w)\big)$	(`glColorTable`)
$\|\bar{V}\|_{k=1,\ldots,\infty}$	(`glHistogram`)

  At this time all results were clamped to $[0, 1]$, which was particularly annoying for the addition (values would saturate at 1) and subtraction (negative results evaluated to 0). The powerful `glColorTable` and `glHistogram` operations were unfortunately not well supported in graphics hardware. The computation of vector norms from a histogram $H : \{0, \ldots, 2^m - 1\} \to \mathbb{N}$, which assigns the number of appearances in $\bar{V}$ to every value of the number format, can be performed as

$$\|\bar{V}\|_k = \left(\sum_{y=0}^{2^m-1} \left(r^{-1}(y)\right)^k \cdot H(y) \right)^{\frac{1}{k}}, \tag{4.1}$$

  for $k = 1, 2, \ldots$. For $k = \infty$ we simply pick up the largest $|r^{-1}(y)|$ with $H(y) > 0$, where r^{-1} is the inverse transformation from the encoded values to the represented numbers (cf. Table 4.2 on page 123).

- DX7 GPUs, 1999-2000.

  $\bar{V}_\alpha + \bar{W}_\alpha$ (`GL_EXT_texture_env_combine`)

  $\bar{V}_\alpha + \bar{W}_\alpha - \frac{1}{2}$ (`GL_EXT_texture_env_combine`)

  $4 \cdot \mathrm{dot3}(\bar{V}_\alpha.xyz - \frac{1}{2}, \bar{W}_\alpha.xyz - \frac{1}{2})$ (`GL_EXT_texture_env_dot3`)

  $(\bar{U}_\alpha < \frac{1}{2})?\bar{V}_\alpha : \bar{W}_\alpha$ (`GL_NV_register_combiners`)

  Here dot3(.,.) denotes a dot product of 3-vectors. While DX6 GPUs offered support for two textures, DX7 GPUs extended this to four, each with an individual texture environment which was able to perform many consecutive operations (`GL_EXT_texture_env_combine`). The `GL_NV_register_combiners` extension offered similar arithmetic operations, a conditional statement and additional input and output mappings. It introduced the signed 9 bit format, but only internally.

- DX8 (VS1, PS1) GPUs, 2001-2002.

  $f(\bar{V}_\alpha.x, \bar{V}_\alpha.y), f(\alpha_x + \bar{V}_\alpha.x, \alpha_y + \bar{V}_\alpha.y)$ (`GL_NV_texture_shader`,

  $+, -, \cdot$, mul-add, lerp, dot3/4, `GL_NV_register_combiners`,

  $(. < \frac{1}{2})?, (. >= 0)?$ `GL_ATI_fragment_shader`)

  The `GL_ATI_fragment_shader` extension offers up to 16 instructions for computation and texture access. The `GL_NV_register_combiners` extension executes up to 8 instructions and `GL_NV_texture_shader` up to 4 different texture access modes, some of which perform implicit computations as well. So the extensions are similar in functionality. The `GL_ATI_fragment_shader` has more flexibility in dependent texture accesses and a larger number range $[-8, 8]$, while specific access modes of `GL_NV_texture_shader` can save on computations and the combiners have a more general computing model ($\bar{A}_\alpha \cdot \bar{B}_\alpha + \bar{C}_\alpha \cdot D_\alpha$), though only in $[-1, 1]$. The number of textures units increased to 8.

- DX9 (VS2, PS2) GPUs, 2002-2004.

  assembly language: (`GL_ARB_fragment_program`)

  arithmetic, reciprocal, trigonometric functions, conditional assigments

  The PS2 model together with the introduction of floating point number formats has moved GPU processing to a very general programming level. The limits are now set by the control features of the assembly language rather than the insufficiency in mathematical operations. Moreover, graphics HLLs (Cg [NVIDIA, 2002], GLSL [Ope, 2004]) give also a much easier access to the functionality. The number of textures units is set to 16.

- DX9+ (VS2-VS3, PS2-PS3) GPUs, 2004.

  extended assembly language: (`GL_NV_fragment_program2`)

  sub-routines and restricted forms of dynamic branching, looping, variable indexing

  With the DX9 GPUs the desire for arithmetic functionality has been basically fulfilled.

The new generation improves on the length of the programs and removes restrictions of multiple dependent texture accesses. But the development focus has changed towards evolution of flow control in the assembly programs and more flexibility in the large scale data-flow within the graphics pipeline.

Before turning to the concrete implementations we should emphasize some developments which may not be so clear from the above presentation and the previous discussions.

- The FP has quickly prevailed over fragment blending.

  Already the DX6 GPUs offered the combination of two textures in one pass. In the beginning the number of operations which could be applied at this stage was smaller than the number of blending modes, but the texture environments were evolving quickly. With the DX7 GPUs there were already four texture environments and thus more computational power than in the blending. Also more different values could be combined at once, e.g. the four neighboring node values. Last but not least, many architectures have a higher texture than pixel fill rate, because each pixel pipeline has several texture mapping units (TMUs).

- Before DX8 GPU the use of non-linear functions was usually slow.

  The fast dependent texture access was a major breakthrough for the evaluation of non-linear functions. The previous alternatives (`glPixelMap, glColorTable, GL_SGI_texture_color_table, GL_SGIS_pixel_texture`) were either slow or not supported on PC GPUs.

- In DX9 GPUs the fixed pipeline is sometimes faster than the programmable.

  If the functionality required is available in the fixed, parameter controlled pipeline than the chances are good that it executes faster than the an equivalent assembly program for the programmable FP. This is the case if the chip contains pipelined hardwired functionality to perform the task, which is usually faster than the instruction execution in the FP. This may change in future, since the PEs of the FP are growing both in breadth and depth. Also, more and more code for the hardwired parts is translated by the graphics driver into the assembly language of the programmable PEs. Thus, transistors are saved and the percentage of programmable PEs in GPUs increases.

4.1.3 Level-Set Segmentation

We present the implementation of a solver for the *level-set equation* used for image segmentation. The continuous model is discussed in Section 2.1.2 on page 19, the discrete and quantized model in Section 2.4 on page 55. This implementation poses very weak requirements on the graphics hardware (a subset of DX6) and thus would run on almost any GPU. The following discussion is based on the publication [Rumpf and Strzodka, 2001a].

As discussed in Section 2.4 on page 55 we implement the level set equation with external forces exclusively. This work was later extended by other researchers [Lefohn et al., 2003, 2004] to include curvature terms (cf. Section 2.1.2 on page 19) and an adaptive computing scheme on newer graphics hardware.

4.1.3.1 Implementation

We recall the quantized upwind scheme for the *level-set equation*, which needs to be implemented (Eq. 2.73 on page 57):

$$
\begin{aligned}
\bar\Phi^{n+1}_\alpha &= \bar\Phi^n_\alpha \ominus \bar g^{\mathrm{lin}}_\alpha(D^-\bar\Phi^n, D^+\bar\Phi^n)\\
g^{\mathrm{lin}}_\alpha(U,V) &:= (\tfrac{\tau^n}{h}\bar F)^+_\alpha \odot \|(\bar U^+_\alpha, \bar V^-_\alpha)\|_{\mathrm{lin}} \oplus (\tfrac{\tau^n}{h}\bar F)^-_\alpha \odot \|(\bar U^-_\alpha, \bar V^+_\alpha)\|_{\mathrm{lin}}\\
\|X\|_{\mathrm{lin}} &:= c\odot\|X\|_1 \oplus (1-c)\odot\|X\|_\infty\\
D^+_\alpha\bar\Phi^n &:= \begin{pmatrix}\bar\Phi^n_{\alpha+(0,1)}\ominus\bar\Phi^n_\alpha\\ \bar\Phi^n_{\alpha+(1,0)}\ominus\bar\Phi^n_\alpha\end{pmatrix}\\
D^-_\alpha\bar\Phi^n &:= \begin{pmatrix}\bar\Phi^n_\alpha\ominus\bar\Phi^n_{\alpha-(0,1)}\\ \bar\Phi^n_\alpha\ominus\bar\Phi^n_{\alpha-(1,0)}\end{pmatrix},
\end{aligned}
\tag{4.2}
$$

with the quantized operations $\{\oplus, \ominus, \odot\}$ corresponding to their natural analogs $\{+, -, \cdot\}$ (Section 2.2.3.1 on page 36).

The speed function $\bar F$ is the discretization of

$$
f(x) = c(x) + g_1(p(x)) + g_2(\|\nabla p(x)\|)\,,
\tag{4.3}
$$

which depends on the image intensities $p(x)$ and its gradient modulus $\|\nabla p(x)\|$. The functions c, g_1, g_2 and their parameters are set by the user (cf. Section 2.1.2.1 on page 19). The initial level-set $\bar\Phi^0$ function is interactively generated by the user by specifying some points in the image as the starting points of the level-set evolution (cf. Figure 4.4 on page 133).

The scheme has been designed such that small and large factors compensate and the number range $[0,1]$ can be fully utilized by all involved variables. Moreover, we can exploit all the available precision because all results can be expressed by storing the positive and negative results separately without the need for a signed number format, e.g. for the precomputed velocities we reserve two textures $(\frac{\tau^n}{h}\bar F)^+_\alpha$ and $(\frac{\tau^n}{h}\bar F)^-_\alpha$. In particular, the differences $D^\pm\bar\Phi^n$ are only needed either as the positive $\left(D^\pm\bar\Phi^n\right)^+$ or negative $\left(D^\pm\bar\Phi^n\right)^-$ part, which corresponds directly to the subtraction with the implicit clamping to $[0,1]$ in the *fragment blending* functionality (Section 4.1.2.3 on page 124). So the required operations needed to implement Eq. 4.2 are:

operation	formula	fragment blending
multiplication	$\bar{V}_\alpha \cdot \bar{W}_\alpha$	`glBlendFunc`
scalar factor	$a\bar{V}_\alpha$	`glBlendFunc`
addition	$\bar{V}_\alpha + \bar{W}_\alpha$	`glBlendFunc`
subtraction	$(\bar{V}_\alpha - \bar{W}_\alpha)^+$	`glBlendEquation`
maximum	$\max(\bar{V}_\alpha, \bar{W}_\alpha)$	`glBlendEquation`
index shift	$\bar{V}_{\alpha+\gamma}$	`glVertex`

The last operation, the index-shift, needed for the differences, is simply achieved by the change of the drawing position for the desired image. The binary operations with fragment blending are performed in the following way. The first operand is displayed into the color buffer (cf. Section 4.1.1.2 on page 112). Then the setting of source and destination factors and the blending equation determine in which manner the following image will be combined with the source. Rendering the second operand into the buffer thus performs the desired operation. The result can be further processed by another operation by copying withing the color buffer or after reading it to a texture. We implemented both types of data-flow, i.e. copy-to-frame-buffer and copy-to-texture (Section 4.1.2.1 on page 119). Algorithm 4.1 outlines the overall program execution in pseudo code notation.

Algorithm 4.1 Algorithm for the level-set segmentation in graphics hardware (Eq. 4.2 on the preceding page). The involved operations use only very basic graphics hardware functionality (a subset of DX6) and could be run on almost any GPU.

```
level set segmentation {
  load the original image $\bar{P}$;
  compute the initial function $\bar{\Phi}^0$ from the user defined seed points;
  compute the velocities $\frac{\tau^n}{h}\bar{F}^+$ and $\frac{\tau^n}{h}\bar{F}^-$ from $\bar{P}$, $\|\nabla\bar{P}\|$ and user specified parameters;
  initialize the graphics hardware with $\frac{\tau^n}{h}\bar{F}^+$,$\frac{\tau^n}{h}\bar{F}^-$ and $\bar{\Phi}^0$;
  for each time-step $n$ {
    calculate the differences $\bar{U}^+ = (D^-\bar{\Phi}^n)^+$ and $\bar{V}^- = (D^+\bar{\Phi}^n)^-$;
    approximate the Euclidean norm by $\|(\bar{U}^+, \bar{V}^-)\|_{\text{lin}}$;
    compose the first flux addend $(\frac{\tau^n}{h}\bar{F})^+ \cdot \|(\bar{U}^+, \bar{V}^-)\|_{\text{lin}}$;
    calculate the differences $\bar{U}^- = (D^-\bar{\Phi}^n)^-$ and $\bar{V}^+ = (D^+\bar{\Phi}^n)^+$;
    approximate the Euclidean norm by $\|(\bar{U}^-, \bar{V}^+)\|_{\text{lin}}$;
    compose the second flux addend $(\frac{\tau^n}{h}\bar{F})^- \cdot \|(\bar{U}^-, \bar{V}^+)\|_{\text{lin}}$;
    sum the flux addendens to the full flux $\bar{g}^{\text{lin}}(D^-\bar{\Phi}^n, D^+\bar{\Phi}^n)$;
    update the level-set function $\bar{\Phi}^{n+1} = \bar{\Phi}^n - \bar{g}^{\text{lin}}(D^-\bar{\Phi}^n, D^+\bar{\Phi}^n)$;
  }
}
```

If the flux $\bar{g}^{\mathrm{lin}}(D^{-}\bar{\Phi}^{n}, D^{+}\bar{\Phi}^{n})$ becomes very small or vanishes altogether, which can be checked by computing a norm of this vector (Eq. 4.1 on page 125), then a simple rescaling of $\bar{\Phi}^{n}$ and/or the flux addends will continue the evolution until the velocities are really zero (Section 2.4.3 on page 58). In practice, however, the `glHistogram` functionality used for the evaluation of the vector norms performed poorly. So at this time it was unavoidable that the user himself would envoke the scaling if he wanted the curve evolution to continue. This is was not a huge disadvantage, since interactivity with the user, who had to set the starting points and the velocity parameters was assumed from the beginning.

4.1.3.2 Results

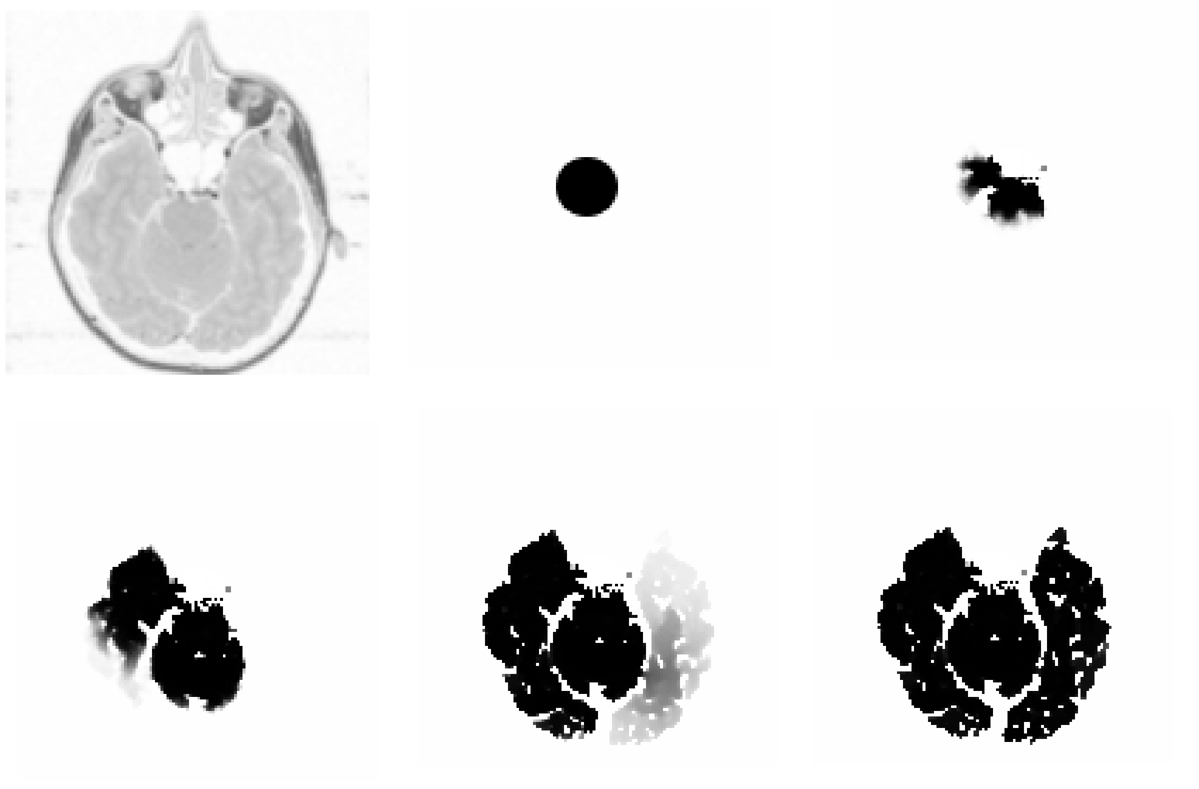
The computations had been performed on the InfiniteReality2 graphics system of the SGI Onyx2 4x195MHz R10000 and a PC graphics card powered by NVIDIA's GeForce2 Ultra 250MHz(4x2) / 230MHz(128 bit DDR) chip with a precision of 12 and 8 bit respectively. Apart from the pre-calculation of the user controlled speed function $\frac{\tau^{n}}{h}\bar{F}$ and the initial level-set function $\bar{\Phi}^{0}$, all computations took place in the graphics system. In the definition of $\bar{F}$ (Eq. 4.3 on page 128), we had usually used no constant velocity ($c(x) = 0$), a parameter controlled polynomial for $g_1(x)$ and a *Perona-Malik function* for $g_2(x)$.

Figure 4.2 on the next page shows the segmentation on a slice through the human brain. The images closely resemble software results. Moreover, the first two images of the sequence demonstrate that the allowance of negative values in the speed function enables the initially too large contours to withdraw from regions with unfitting intensities. This property requires the level-set formulation and cannot be implemented with the fast marching method [Sethian, 1999]. We also notice that the applied approximation of the Euclidean norm is not visible in real world applications. Only if we observe a curve evolution with a constant velocity the approximative structure comes into sight (Figure 4.3 on page 132).

In Figure 4.4 on page 133 several differently colored seed points evolve independently to segment the pickets of a barbed wired fence. In this example all active contours use the same velocities, but in general the color components may evolve independently along different velocities while still being encoded in a single image, because the graphics hardware operates internally on 4-vectors (RGBA) anyway.

Iintially we implemented the copy-to-frame-buffer data-flow (cf. Section 4.1.2.1 on page 119), as it avoids unneccessary data copying when working with the fragment blending operations exclusively. However, copying turned out to be fairly slow on the SGI Onyx2 with enabled blending modes. In late 2000 the same applied to the GeForce2. But the graphics drivers for the GeForce2 started to offer a fast copy-to-texture at this time. Although the copy-to-texture involves additional unnecessary data movement as opposed to copy-to-frame-buffer and it required the textures to be at least (RGB) as the color buffer, it turned out to be still much faster. The execution time for one time-step dropped by more than an order of magnitude from 30ms to 2ms for a 128^2 image, which corresponds approximately to 65% (128^2 pixel * 80 data transfers / 2ms = 655 Mpixel/s) of the theoretic maximal throughput of 1000 Mpixel/s (4

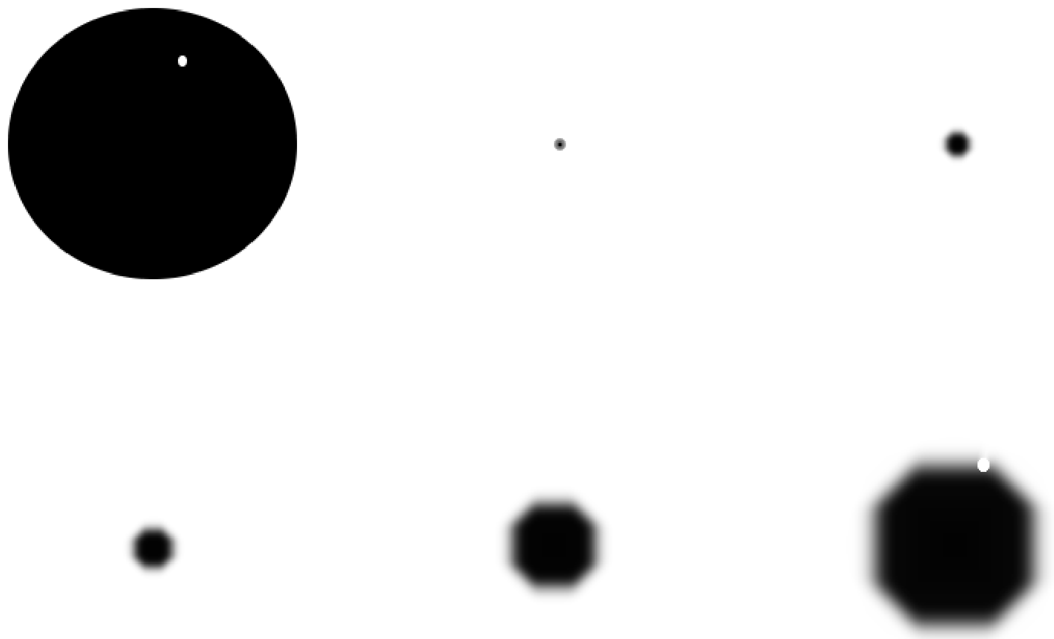
Figure 4.2 Segmentation of a human brain computed in DX7 graphics hardware on a 128^2 image resolved by 8 bit. Besides the original image, the timesteps 0, 10, 50, 150 and 350 are depicted. The computation of one timestep took 2ms.



pipelines * 250MHz). The overhead is caused by the frequent implicit flushing of the graphics pipeline, happening each time before the content of the frame-buffer can be copied back to a texture. In view of this complications, the 65% of the unreachable peak perfromance were a very efficient utilization of hardware resource. One should note that this implementation was designed for the fragment blending capabilities, at a time when subtraction in the FP was unavailable. The recourse to the copy-to-texture in the GeForce2 was a late remedy for the continued lack of a fast copy-to-frame-buffer functionality. The extension of this work by Lefohn et al. is centered around the more powerful DX8 Fragment Processor (FP) [Lefohn et al., 2003] and thus relates performance to the texture fill rate rather than the pixel fill rate.

Since the fast copy-to-texture enforced the use of RGB or RGBA textures one could also make use of the extra colors. The parallel evolution of several level-sets could thus be performed at almost ($< 3\%$) no extra cost (Figure 4.4 on page 133), as along as sufficient bandwidth was available, which was the case for the GeForce2 Ultra.

Figure 4.3 The visible effect of the Euclidean norm approximation during segmentation in totally homogenous areas. The circle with the little hole on the upper left is the region to be segmented. The subsequent images show the expansion of a small circle under the approximate norm $\|.\|_{\text{lin}}$. As expected from the form of the unit sphere of the norm (Figure 2.2 on page 57) the circle evolves into a octagon. This effect is only visible in totally homogenous areas, where the entire front expands with the same velocity.

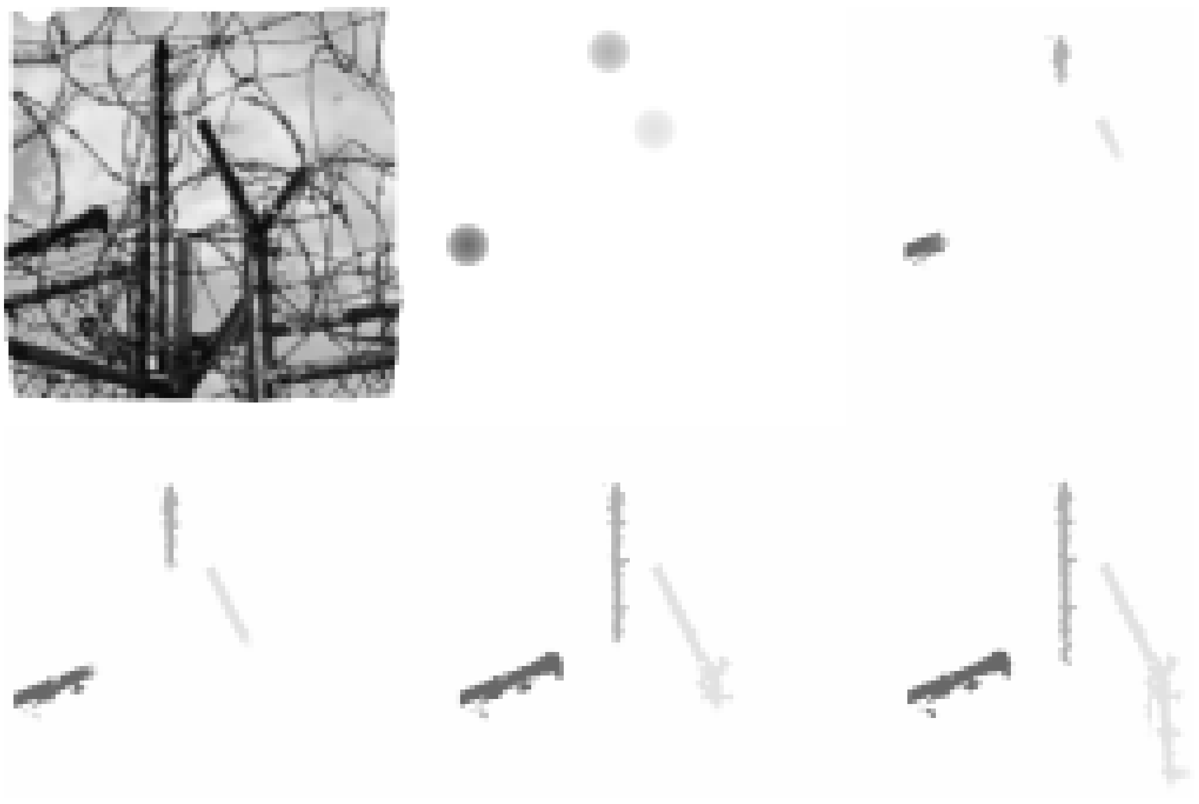


4.1.4 Anisotropic Diffusion

We discuss the implementation of explicit and implicit solvers for the *anisotropic diffusion* model for image denoising. This includes the isotropic *non-linear diffusion* model as a special case. The continuous model is discussed in Section 2.1.1 on page 15, the discrete, quantized model in Section 2.3 on page 43. The graphics hardware implementation of the diffusion models has been previously presented in the publications [Diewald et al., 2001; Rumpf and Strzodka, 2001b,c].

Ancestor to this work was the implementation of Gaussian filtering in [Hopf and Ertl, 1999], which corresponds to the solution of the *linear diffusion* problem. Hopf and Ertl also pioneered other hardware accelerated filter applications, such as wavelet transformations [Hopf and Ertl, 2000b] and morphological analysis [Hopf and Ertl, 2000a]. Because diffusion is a key ingredient in many physical processes, many successors considered its implementation as

Figure 4.4 Parallel segmentation of fence pickets in DX6 graphics hardware.



part of their solvers, e.g. [Harris et al., 2002; Kim and Lin, 2003]. In [Colantoni et al., 2003] anisotropic diffusion has been addressed explicitly again. Linear algebra with sparse matrices and implicit solvers in general have later also been discussed in [Bolz et al., 2003; Goodnight et al., 2003; Harris et al., 2003; Krueger and Westermann, 2003].

The succeeding papers differ from the following presentation in that they assume newer graphics hardware with floating point number formats (DX9). Only [Harris, 2002] examines the effects of roundoff error in fixed point diffusion under a local *white-noise model* assumption. But neither floating point numbers nor good local error bounds (which are unattainable for 8 bit anyway) are necessary for the preservation of the qualitative behavior of the continuous diffusion models over a long time, as demonstrated by the quantized scale space (Section 2.3.3 on page 49) generated by our schemes.

4.1.4.1 Implementation

The quantized explicit mass preserving scheme (Eq. 2.48 on page 48) to be implemented reads

$$\bar{U}^{n+1} = A^{-}[\bar{U}^{n}_{\sigma}] \boxdot \bar{U}^{n} \tag{4.4}$$

$$= \bar{U}^n \ominus \left(\frac{\tau^n}{h^2} L[\bar{U}^n_\sigma]\right) \boxdot \bar{U}^n \,,$$

with the quantized operations $\{\oplus, \ominus, \odot\}$ from Section 2.2.3.1 on page 36, and the quantized *mass-exact matrix vector product* denoted by $\boxdot$ (Eq. 2.29 on page 40). In case of the semi-implicit scheme we have to solve

(4.5)
$$\begin{aligned} A^+[\bar{U}^n_\sigma] \cdot \bar{U}^{n+1} &= \bar{U}^n \\ A^+[\bar{U}^n_\sigma] &:= \mathbb{1} + \frac{\tau^n}{h^2} L[\bar{U}^n_\sigma] \,, \end{aligned}$$

with a quantized version of the iterative solvers (Eqs. 2.39, 2.40 on page 45).

The components of the stiffness matrix $L[\bar{U}^n_\sigma]_{\alpha\beta}$ are given as a weighted sum of integrations over the elements adjacent to the α node (cf. Figure 2.1 on page 27):

(4.6)
$$\begin{aligned} L[\bar{U}^n_\sigma]_{\alpha\beta} &= \sum_{E\in E(\alpha)} \sum_{i,j\in\{x,y\}} (G^n_E)_{i,j} (S^{\alpha\beta}_E)_{i,j} \\ G^n_E &:= G(\nabla U^n_\sigma(m_E)) \\ (S^{\alpha\beta}_E)_{i,j} &:= (\partial_i \Phi_\alpha, \partial_j \Phi_\beta)|_E \,, \end{aligned}$$

where $(G^n_E)_{i,j}$ are the components of the *diffusion tensor* (Eq. 2.44 on page 46) and $(S^{\alpha\beta}_E)_{i,j}$ pre-integrated constants depending on the Finite Element (FE) basis functions. On an equidistant mesh with linear FEs we have an explicit formula for the inner sum which is used for the computation (Eq. 2.45 on page 46). In case of the isotropic *non-linear diffusion* model, where the weight function $\tilde{g}$ is scalar, the local formula is much simpler:

(4.7)
$$\begin{aligned} L'[\bar{U}^n_\sigma]_{\alpha\beta} &= \sum_{E\in E(\alpha)} G'^n_E S'^{\alpha\beta}_E \\ G'^n_E &:= \tilde{g}(\nabla U^n_\sigma(m_E)) \\ S'^{\alpha\beta}_E &:= (\nabla\Phi_\alpha, \nabla\Phi_\beta)|_E \,. \end{aligned}$$

Because the formulas mix positive and negative numbers at various stages the separate treatment of them would have caused a large overhead. Instead, we emulate a signed format in the unsigned 8 bit format. Our first approach was to reserve the range $[-2, 2]$ for computations and apply the formulas in Table 4.2 on page 123 to obtain the correct results. With the availability of register combiners which support a signed 9 bit format internally, we could easily map $[0, 1]$ to $[-\frac{1}{2}, \frac{1}{2})$ before each computation and work with the signed numbers (see Section 4.1.2.2 on page 121 for details). We remind that the quantized solution satisfies a extremum principle (Eq. 2.53 on page 51), so that we only have to avoid a *saturation* in intermediate computations.

The matrix vector product with the stiffness matrix $\sum_\beta L[\bar{U}^n_\sigma]_{\alpha\beta} \cdot \bar{X}_\beta$ is the computationally most demanding part for both the explicit (Eq. 4.4 on the preceding page) and implicit scheme (Eq. 4.5), since the iterative linear equation system solvers (Eqs. 2.39, 2.40 on page 45) also contain the matrix vector product as their main component. Even the scalar product encountered in the conjugate gradient solver (Eq. 2.40 on page 45) is just a component-wise multiplication followed by the computation of the $\|.\|_1$ norm as described in Eq. 4.1 on page 125.

So these more elaborate schemes are just a quantative increase in requirements compared to the solution of the *level-set equation* in the previous section. From the computational point of view the same arithmetic operations are involved: addition, subtraction, multiplication, vector norms. There is, however, one exception, namely the non-linear dependence of the weights $(G_E^n)_{i,j}$ on the gradient $\nabla U_\sigma^n(m_E)$. For the level-set equation we had also to deal with the non-linear Euclidean norm of a 4-vector and used a linear approximation for it (Eq. 2.72 on page 56). For the reciprocal of diagonal matrix elements in the Jacobi solver (Eq. 2.49 on page 48) and the scalar weight function $\tilde{g}(\|\nabla u_\sigma\|)$ in the non-linear diffusion model, typically given by the *Perona-Malik function* (Eq. 2.3 on page 16), also a simple linear approximation can be used (Figure 4.5 on the following page). This is, however, much more demanding in the anisotropic model for the normalized gradient components (cf. Eq. 2.44 on page 46)

$$b_x^E := \frac{\partial_x U_\sigma^n(m_E)}{\|\nabla U_\sigma^n(m_E)\|}, \qquad b_y^E := \frac{\partial_y U_\sigma^n(m_E)}{\|\nabla U_\sigma^n(m_E)\|}, \tag{4.8}$$

which are two-dimensional non-linear functions:

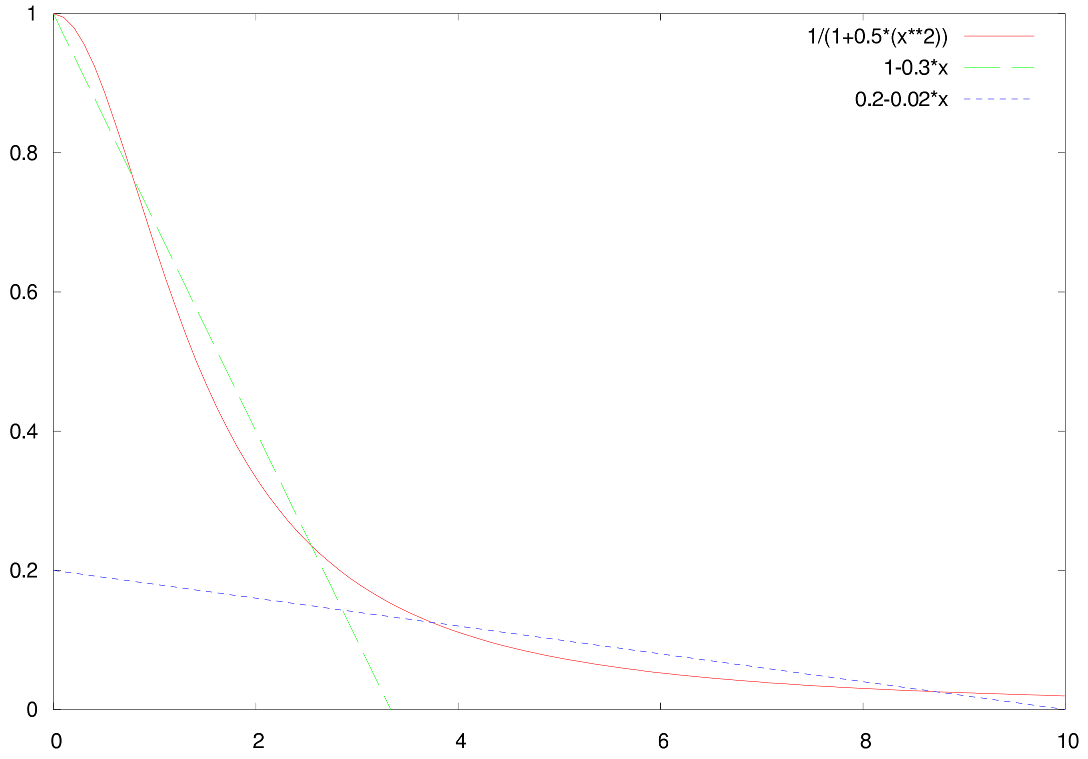
$$f(x,y) \quad := \quad \frac{x}{\sqrt{x^2+y^2}} \quad = \quad \frac{1}{\sqrt{1+(y/x)^2}}.$$

We should recall that there is no division operation in graphics hardware. Even the DX9+ GPUs compute a reciprocal and multiply with the numerator. This is feasible with *floating point numbers*, but in *fixed point*, the reciprocal cannot be expressed appropriately, so that even with a one dimensional lookup table the function would be difficult to evaluate, because only a small part of the codomain of $\frac{1}{\sqrt{x^2+y^2}}$ or $\frac{1}{x}$ could be represented. Only a lengthy logarithmic representation could capture a larger codomain: $f(x,y) = \operatorname{sgn}(x) \cdot \exp\left(\ln|x| - \frac{1}{2}\ln(x^2+y^2)\right)$. But this would be the wrong strategy for fixed point numbers, which represent smaller values with increasingly less binary positions and thus precision. If one has to use an approximation the best option is to resolve the larger values as good as possible say the range $[\frac{1}{4}, 1]$ and disregard everything below. For very small differences (almost homogenous areas) it is not so important to evaluate the anisotropic term in Eq. 2.45 on page 46, since the non-linear diffusion term will diffuse the area anyway.

The dependent texture access introduced with DX8 GPUs solved the above problems in general. One can encode the values of the non-linear function in a 2D texture and use the function variables as coordinates into that texture, thus implementing any two dimensional function (cf. Section 4.1.2.3 on page 124).

The data-flow for the diffusion models uses the copy-to-texture method (Section 4.1.2.1 on page 119), because the multiple texture environments in the FP allow to compute many more results in a single pass. Especially the summations in the frequent matrix vector products with the stiffness matrix (Eqs. 4.6, 4.7 on the preceding page) lend themselves to the processing

Figure 4.5 Approximation of the *Perona-Malik function* by linear functions. If the evalutation of non-linear functions in the hardware architecture is not available, they can be approximated by linear ones. Computing the maximum of the above linear functions gives a good approximation without the need of explicit conditional statements. Clearly, general non-linearities can be very complicated and require many linear segments for a reasonable approximation.



in the FP. There were also no fast alternatives as long as pbuffers were not available. Algorithms 4.2 on the facing page and 4.3 on the next page present the pseudo-code for the explicit and implicit solver respectively.

4.1.4.2 Results

Different versions of the solvers were run on the SGI Onyx2 4x195MHz R10000 with InfiniteReality2 graphics, NVIDIA GeForce2 Ultra 250MHz(4x2) / 230MHz(128 bit DDR) and GeForce3 200MHz(4x2) / 230MHz(128 bit DDR). The Onyx2 offered 12 bit precision the GeForce chips 8 bit. In the non-linear case we used the *Perona-Malik function* for the weight function $\tilde{g}$, in the anisotropic case a constant for g_1 and a combination of a constant and Perona-Malik function for g_2. In the explicit case the time-step width has to comply with $\frac{\tau^n}{h^2} < \frac{3}{16}$ (Eq. 2.56 on page 51) to secure the properties of the quantized scale-space. For the semi-implicit model the time-step width is theoretically not restricted, but concerning the menacing underflow in the iterative schemes (cf. Eq. 2.50 on page 48) we kept $\frac{\tau^n}{h^2}$ below 8.

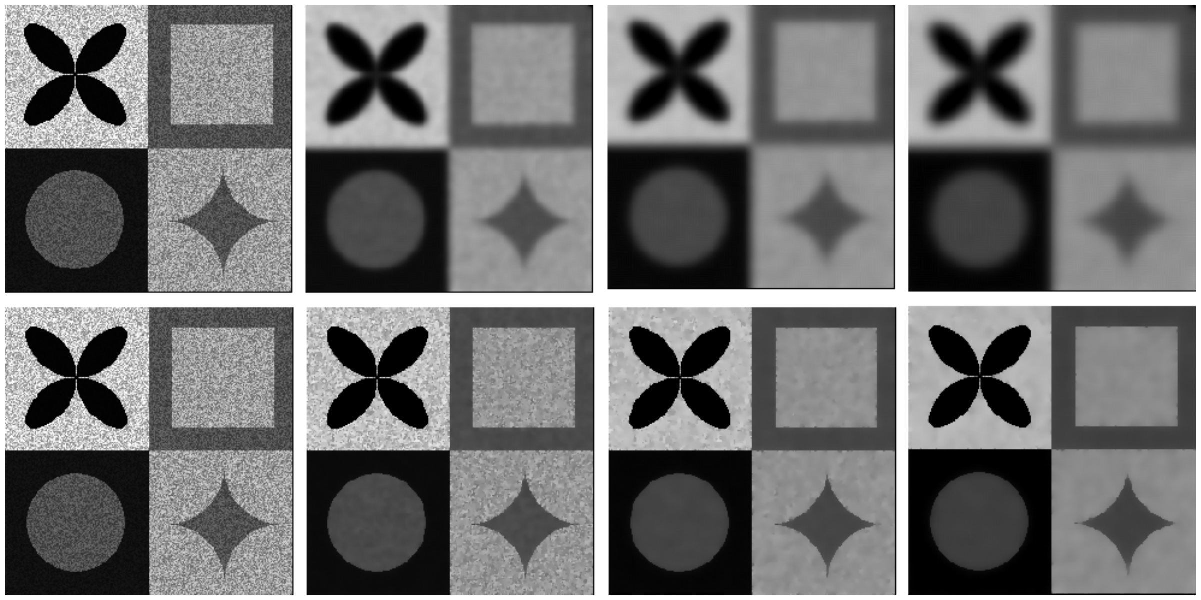
Algorithm 4.2 Explicit scheme for image denoising by anisotropic diffusion implemented in graphics hardware (Eq. 4.4 on page 133). The mass-exact matrix vector product guarantees that the overall mass stays the same, i.e. $\sum_\alpha \bar{U}^{n+1}_\alpha = \sum_\alpha \bar{U}^n_\alpha$, despite quantization and roundoff errors. The scheme generates a quantized scale-space with qualitative properties very similar to the continuous scale-space (Section 2.3.3 on page 49).

```
anisotropic diffusion {
  load the original image $\bar{U}^0$;
  initialize the graphics hardware with $\bar{U}^0$;
  for each timestep $n$ {
    mollify the gradient $\nabla U^n_\sigma$ with a Gaussian kernel;
    calculate/approximate the weights $g^E_1, g^E_2$ and the directions $b^E_x, b^E_y$ (Eq. 2.44 on page 46);
    assemble the stiffness matrix $L[\bar{U}^n_\sigma]$;
    compute the mass-exact matrix vector product $\left(\frac{\tau^n}{h^2}L[\bar{U}^n_\sigma]\right) \boxdot \bar{U}^n$;
    update the solution $\bar{U}^{n+1} = \bar{U}^n \ominus \left(\frac{\tau^n}{h^2}L[\bar{U}^n_\sigma]\right) \boxdot \bar{U}^n$;
  }
}
```

Algorithm 4.3 Semi-implicit scheme for image denoising by anisotropic diffusion implemented in graphics hardware (Eq. 4.5 on page 134). Each step of the iterative linear equation solver involves at least one matrix vector multiplication and several other operations (Eqs. 2.39, 2.40 on page 45). The iterative solver cannot exactly preserve the mass of the vector, but an additional step can correct the mass defect (Eq. 2.52 on page 49).

```
anisotropic diffusion {
  load the original image $\bar{U}^0$;
  initialize the graphics hardware with $\bar{U}^0$;
  for each timestep $n$ {
    mollify the gradient $\nabla U^n_\sigma$ with a Gaussian kernel;
    calculate/approximate the weights $g^E_1, g^E_2$ and the directions $b^E_x, b^E_y$ (Eq. 2.44 on page 46);
    assemble the stiffness matrix $L[\bar{U}^n_\sigma]$;
    initialize the iterative solver $\bar{R}^n = \bar{U}^n, \bar{X}^0 = \bar{R}^n$;
    for each iteration $l$ {
      calculate a step of the iterative solver $\bar{X}^{l+1} = F(\bar{X}^l)$ with $A^+[\bar{U}^n_\sigma] = \mathbb{1} + \frac{\tau^n}{h^2}L[\bar{U}^n_\sigma]$;
    update the solution $\bar{U}^{n+1} = \bar{X}^{l_{\max}}$
  }
}
```

Figure 4.6 Linear (upper row) and non-linear (lower row) diffusion on a graphics workstation. These earlier results from 2000 demonstrated the potential for PDE solvers on graphics hardware and inspired our further research in this direction.

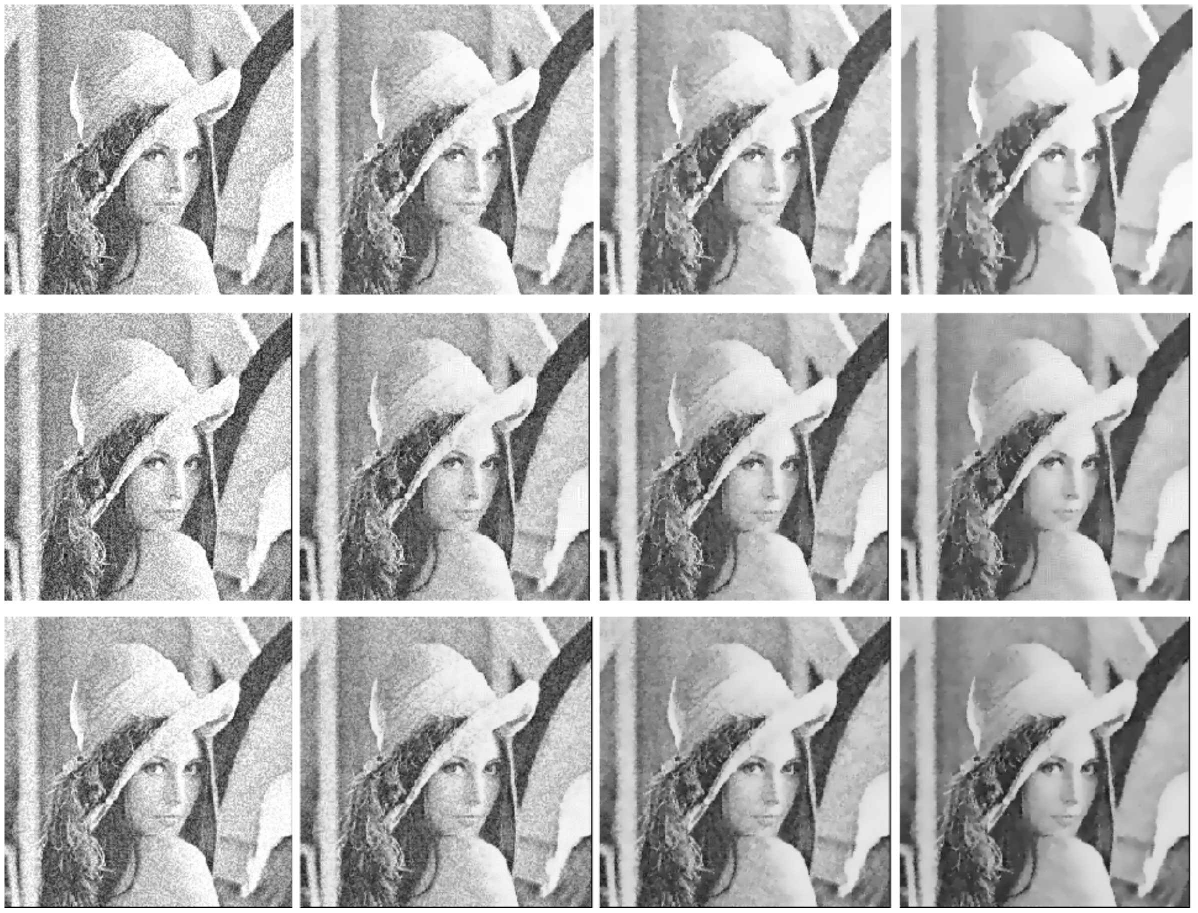


Convolution with a Gaussian kernel, which is equivalent to the application of a linear diffusion model is compared to the results from the non-linear model in Figure 4.6. This test strongly underlines the edge conservation of the non-linear diffusion model and the quantized computation can preserve this property known from the continuous model.

Figure 4.7 on the facing page shows computations with the SGI Onyx2 using the Jacobi and the conjugate gradient solver and compares them to software results. The 12 bit precision sufficies for the task of denoising pictures by non-linear diffusion. But the lack of negative numbers required constant scaling and biasing (see Table 4.2 on page 123). The first implementation did not take explicit care of the resulting roundoff behavior and therefore there is a fine roundoff pattern on the least significant bits in the sequence produced by the Jacobi solver (visible only in the electronic version). The same applies to Figure 4.6 and the upper row in Figure 4.8 on page 140. The subsequent schemes resolved this problem.

At that time many of the involved operations in the graphics pipeline were not supported well in hardware. On average, an iteration of the Jacobi, conjugate gradient solver on 256^2 images took approximately 170ms and 420ms respectively, which was slower than the software solution. The reason for this surprisingly weak performance was easily identified in the unbalanced performance of data transfer between the *frame-buffer* and *video memory*. We have already mentioned that the copy-to-texture machanism (Section 4.1.2.1 on page 119) was not well supported until late 2000. But because after each operation the result had to be copied back to a texture, the access times to the frame-buffer were highly relevant for the overall performance. The discrepancy in performance between writing an image from the video memory

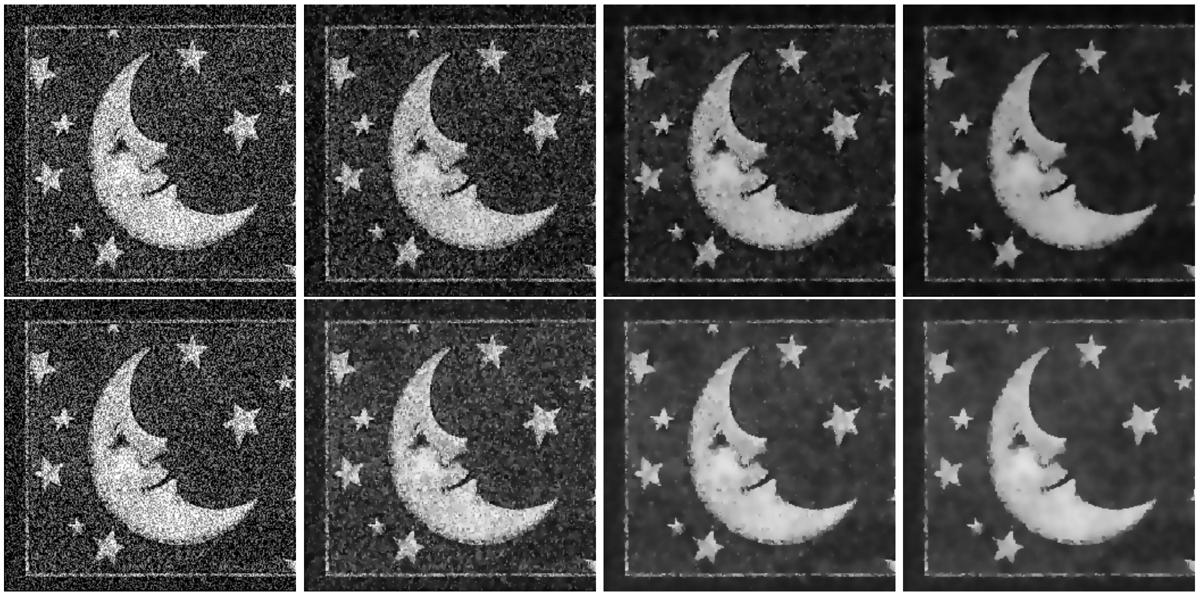
Figure 4.7 Non-linear diffusion solvers in graphics hardware and software; first row: adaptive software pre-conditioned conjugate gradient; second row: Jacobi sover in graphics hardware; third row: conjugate gradient solver in graphics hardware. The black borders are due to restrictions in handling of the boundary conditions in graphics hardware. The fine roundoff pattern in the images (electronic version only) is explained in the text (Section 4.1.4.2 on page 136).



to the frame-buffer and reading it back was as large as 60. The histogram extension used for the computation of the scalar products in the conjugate gradient solver was even worse, explaining the extremely long execution time of the conjugate gradient solver.

Soon after the results from Figure 4.7 were generated [Rumpf and Strzodka, 2001b], the first optimized drivers with a fast copy-to-texture operation for the GeForce series appeared (late 2000). The execution time of the Jacobi solver on the GeForce2 Ultra dropped by more than a factor of 10. This was, however, still less than expected because the lack of a fast dependent texture access as needed for the evaluation of non-linear functions then became the bottelneck. Another problem was the reduced precision of 8 bit on the PC GPUs as opposed to the 12 bit on the graphics workstation. In bright images and for fairly low time-step numbers the mass defect caused by the implicit schemes was hardly visible and could also be counteracted with

Figure 4.8 Mass defect (upper row) and mass exact (lower row) non-linear diffusion in graphics hardware. The used precision is 8 bit. The upper row uses a naive implicit diffusion scheme which causes a mass defect due to many underflows for very small numbers (black areas). The *mass-exact matrix vector product* (Eq. 2.29 on page 40) preserves the overall mass despite low quantization. If the results of the mass defect scheme seem more pleasing, then only because we expect the moon and the stars on a black sky.



scaling strategies (Eq. 2.32 on page 42), though one can notice the effect in Figure 4.7 on the preceding page already. But for badly resolved very low fixed point numbers and many iterations the multiplicative underflows produced a significant mass defect. Figure 4.8 compares this problematic effects with the remedy of the quantized explicit mass preserving scheme (Eq. 4.4 on page 133).

As explained in the previous section, the implementation of the anisotropic model without the availability of two dimensional lookup mechanism for non-linear function evaluation in graphics hardware is very difficult. In 2001 the DX8 GPUs introduced fast dependent texture lookups, which also replaced the rather slow options for one dimensional non-linear functions (cf. end of Section 4.1.2.3 on page 124). Figure 4.9 on the next page compares the first anisotropic diffusion implementation working with linear interpolations with the later one based on dependend texture access. The former had a tendency to further darken dark colors and further brighten bright colors. The reason was the $[-\frac{1}{2}, \frac{1}{2}) \to [0,1]$ encoding which mapped 0 to $\frac{128}{255}$, and the inaccuracy of the approximations for numbers close to 0. This implicit contrast enhancement may seem desirable, but it violates the scale-space approach and destabalizes the evolution in the long run. The DX8 implementation had no such problems. The approximate DX7 version performed at about 20ms on the GeForce2 Ultra, the DX8 version at 8ms on the GeForce3 for one explicit time-step on 256^2 images.

Figure 4.9 Anisotropic diffusion models implemented in DX7 (middle) and DX8 (right) graphics hardware. The images show results after 10 time-steps in the explicit scheme on 128^2 images. The DX7 version had to use linear interpolations and thus produced stable results only for relatively few time-steps. With non-linear two dimensional functions in DX8 these restrictions were eliminated.

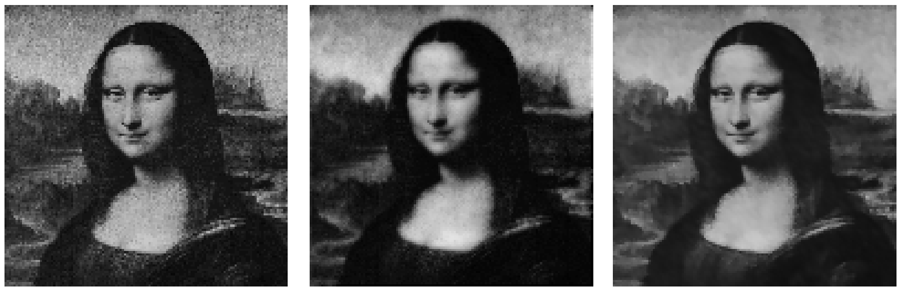
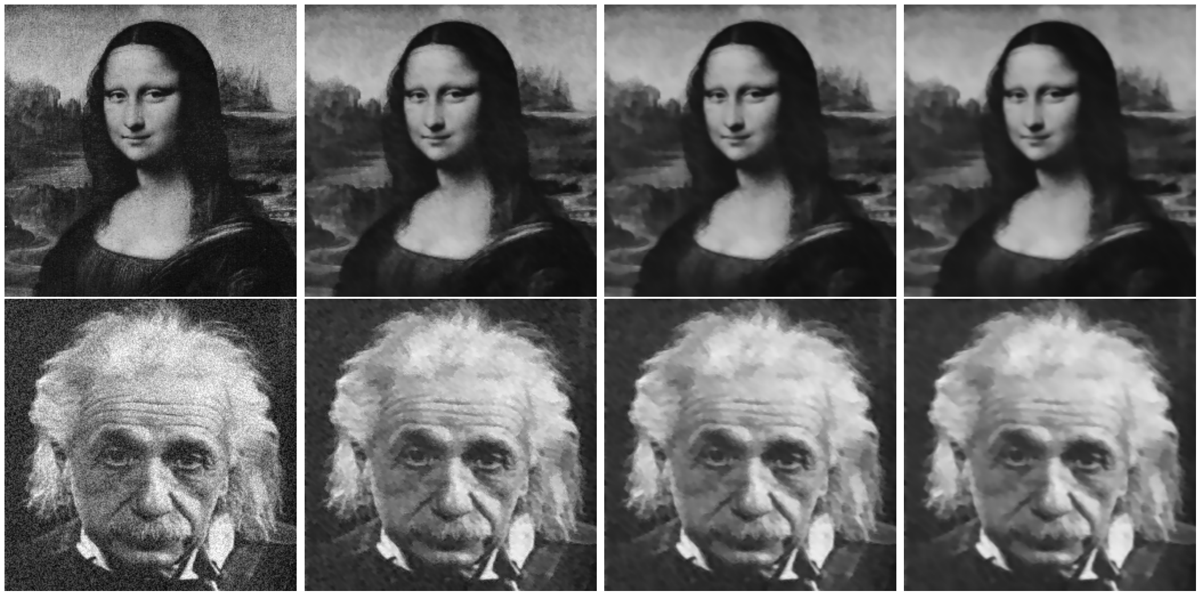


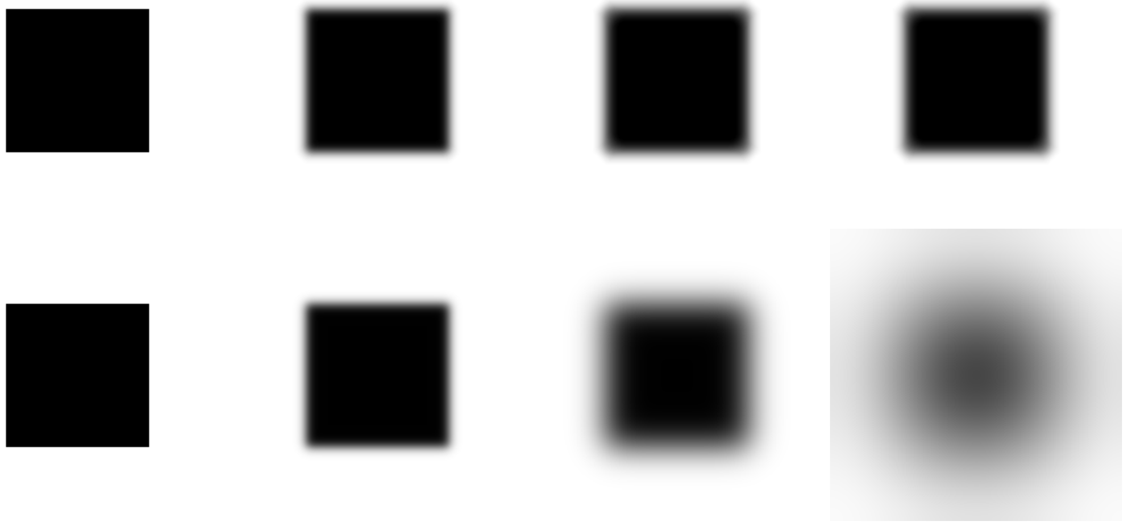
Figure 4.10 Anisotropic diffusion with the virtual signed 16 bit format in DX8 graphics hardware on 256^2 images.



The 16 bit formats introduced with DX8 GPUs were only applicable in special situations and there was still no 16 bit *frame-buffer* format, so that the precision could not be maintained throughout the pipeline. But these GPUs had enough computing resources to emulate a *virtual signed 16 bit format* [Strzodka, 2002] in the 8 bit color channels of RGBA8 textures. Figure 4.10 presents anisotropic diffusion results obtained with this format.

The advantage of the additional bits can be best seen in Figure 4.11 on the next page where we apply the 8 and 16 bit schemes as linear diffusion for a long time. Analysis of the qunatized stop criteria (Eq. 2.66 on page 53) shows that the diffusion process stops roughly when

Figure 4.11 Comparison of the diffusion schemes in 8 bit (upper row) and 16 bit (lower row). A simple linear diffusion is computed with the mass preserving anisotropic scheme to empirically verify its quantized stop criteria (Eq. 2.66 on page 53). The 8 bit diffusion stops quickly due to insufficient precision in intermediate computations. The 16 bit results clearly demonstrate the huge advantage of the additional 8 bit in the *virtual signed 16 bit format* (Section 4.1.2.2 on page 121).



differences between neighbouring values in a grid element become smaller than $8\varepsilon^{\mathcal{Q}}$, where $8\varepsilon^{\mathcal{Q}} = 2^{-5}$ for 8 bit and $8\varepsilon^{\mathcal{Q}} = 2^{-13}$ for 16 bit (cf. Eq. 2.25 on page 35). In other words a smooth ramp from 0 to 1 of $2^5 = 32$ or $2^{13} = 8192$ pixels cannot be further diffused in 8 or 16 bit respectively.

A specific linear diffusion scheme can diffuse larger areas. The above numbers apply to the full anisotropic scheme with the guaranteed mass preservation and no scaling strategies (Section 2.2.3.4 on page 41) applied. They clearly emphasize the importance to have more significant bits in the number representation than in the initial data. They also explain why research on general purpose computations [GPGPU] increased significantly with the introduction of high precision floating point formats in DX9. In many applications not the *exponent* properties of the floating point formats were decisive but a *mantissa* with at least 16 bit (cf. Table 4.3 on page 124). The schemes presented in this and the previous section function on 8 bit only because of a very careful design. With a 16 bit *fixed point number* format many more applications could have been covered. But since 2003 the *floating point number* formats dominate the further development of GPUs, so the next section explores the use of this additional precision in a much more demanding context.

4.1.5 Gradient Flow Registration

This section deals with the image registration performed by a gradient flow PDE. The continuous model is discussed in Section 2.1.3 on page 22, the discrete, quantized model in Section 2.5 on page 59. Results on the graphics hardware implementation of this method appeared previously in [Strzodka et al., 2003, 2004].

Because with DX9 graphics hardware many of the previous restrictions disappeared, a lot of research on general use of GPUs evolved since then. We refer to [GPGPU] for a comprehensive overview. The most related work is that on multi-grid solvers on GPUs also discussed in [Bolz et al., 2003; Goodnight et al., 2003] where they have been applied to fluid dynamics. Here, we use the multi-grid hierarchy for a fast linear equation solver and additionally as an efficient representation of a multi-scale hierarchy, which we use for a robust problem regularization. We also present the implementation of an adaptive time-step control governed by Armijo's rule.

4.1.5.1 Implementation

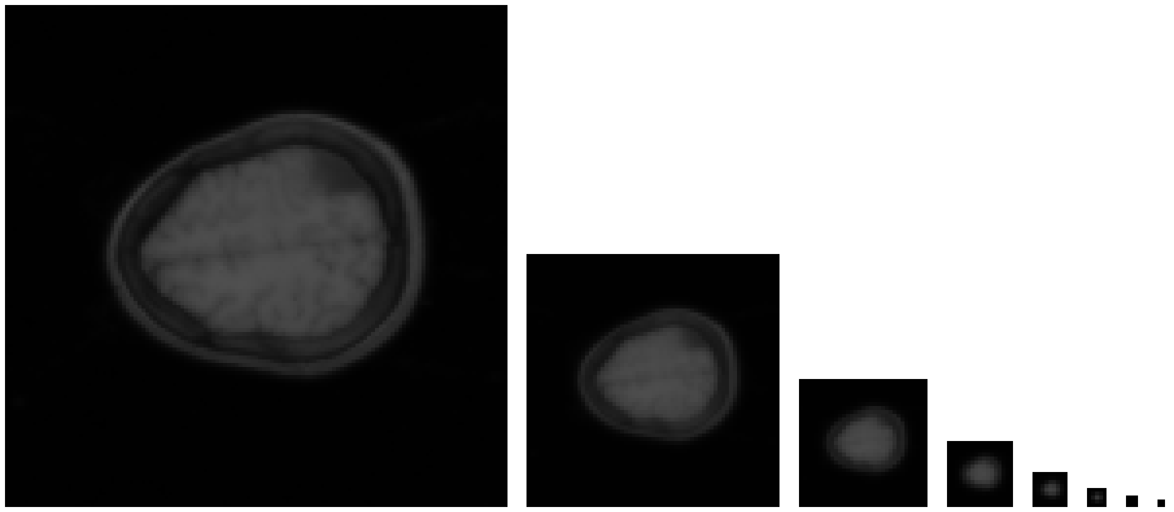
We recall that the discrete, quantized scheme for the image registration consists of three nested loops (Eq. 2.83 on page 61):

$$\begin{aligned}
\bar{U}^{n+1}_{\epsilon_k} &= \bar{U}^n_{\epsilon_k} \ominus \frac{\tau^n_{\epsilon_k}}{h_{l(\epsilon_k)}} \odot \mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}], \qquad n = 0,\dots,N_{\epsilon_k}-1\,, \\
\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}] &= (\bar{T}_{\epsilon_k} \circledcirc (\mathbb{1}+\bar{U}^n_{\epsilon_k}) \ominus \bar{R}_{\epsilon_k}) \odot \left(h_{l(\epsilon)}\nabla_{h_{l(\epsilon_k)}}\bar{T}_{\epsilon_k} \circledcirc (\mathbb{1}+\bar{U}^n_{\epsilon_k})\right)\,, \\
\bar{U}^0_{\epsilon_k} &= \pi^{l(\epsilon_{k+1})}_{l(\epsilon_k)}\left(\bar{U}^{N_{\epsilon_{k+1}}}_{\epsilon_{k+1}}\right), \qquad k = K-1,\dots,0\,, \\
\bar{U}^0_{\epsilon_K} &= \bar{0}\,.
\end{aligned} \tag{4.9}$$

The outer loop with index k runs from the coarse (ϵ_K) to the fine (ϵ_0) scale representations and uses the prolongation operator ($\pi^{l(\epsilon_{k+1})}_{l(\epsilon_k)}$) to transfer data onto finer grids. The middle loop with index n performs the gradient descent on a given scale until the last index N_{ϵ_k} for which the change in data is sufficiently small (Eq. 2.81 on page 61). The inner loop determines for each update (first formula line) the maximal time-step width $\tau^n_{\epsilon_k}$ which satisfies Armijo's rule (Eq. 2.76 on page 60) by maximizing the time-step width τ in Eq. 2.85 on page 62. Naturally, the multi-grid V-cycle (Eq. 2.82 on page 61) performs also an inner loop from the current grid $\Omega_{h_{l(\epsilon)}}$ up to the coarsest grid Ω_{h_L} and back, where Eq. 2.78 on page 60 describes the relation between the grid levels and the scales.

The DX9 graphics hardware allows a high level approach to the graphics functionality. First of all the availability of a floating point format of at least s16e7 precision (cf. Table 4.3 on page 124) eliminates most of the low level implementational concerns about exceeded number ranges in intermediate computations. However, with floating point numbers we must be

Figure 4.12 The multi-grid hierarchy encoded in textures of different spatial resolution. Each grid level Ω_{h_l} serves for the representation of several scales ϵ_k of the multi-scale hierarchy. In contrast to the multi-scale hierarchy, which is a regularization concept of the registration model, the multi-grid hierarchy is an implementational construct for the reduction of computations.



aware of the dramatic errors resulting from interferences of numbers on different scales (Section 2.2.2.1 on page 27). Our quantized scheme does not run into these problems, since we make hardly use of the *exponent* and rather benefit from the large *mantissa*.

An even greater impact on implementations for DX9 graphics hardware has the hardly restricted programmability of the Vertex Processor (VP) and the Fragment Processor (FP). These parts have become so powerful that the other processing elements in the graphics pipeline are hardly used for scientific computations anymore (cf. Figure 4.1 on page 112). The increased functionality is also available at a higher abstraction level with graphics High Level Languages (HLLs) included in the APIs OpenGL and DirectX for the configuration of the VP and FP. We have used Cg [NVIDIA, 2002] for the programming of the configurations. Cg is another graphics HLL with the advantage that it can compile the same code for different APIs and profiles, which express the functionality level of the VP and FP. Thus the following description of the implementation will focus less on the involved arithmetic operations, but more on the higher level of required ***kernels***, i.e. configurations of the programmable VP and FP.

The two dimensional input images T and R are represented as 2D textures on the finest grid Ω_{h_0}. The multi-grid hierarchy $(\Omega_{h_l})_{l=0,\dots,L}$ corresponds to textures of successively smaller size (Figure 4.12). Several such hierarchies are reserved in graphics memory to store any intermediate results. All textures are implemented as floating point *pbuffers* and we use the render-to-texture machanism (Section 4.1.2.1 on page 119), which avoids unnecessary data transfers. Computations are performed by loading a computational kernel to the FP, e.g.

a prolongation kernel, and streaming the texture operands through that kernel into a target pbuffer. Thereby the vertex processor is used to generate the texture coordinates for the access to neighboring nodes in the operand textures. The target pbuffer can then be used as a texture operand in the succeeding operation (cf. Figure 4.1 on page 112).

Algorithm 4.4 Algorithm for image registration in graphics hardware (Eq. 4.9 on page 143). Because of the involved numerical computations, an implementation is only feasible in DX9 or higher functionality graphics hardware.

```
image registration $\{$
  intitialize the graphics hardware with $\bar{T}, \bar{R}$;
  reset the displacement on the coarsest scale $\bar{U}^0_{\epsilon_K} = \bar{0}$;
  perform gradient flow at scale $\epsilon_K$;
  for each scale $\epsilon_k$, $k = K-1, \ldots, 0$ $\{$
    initialize the displacement with the previous scale solution performing
      a prolongation if the grid level also changes $\bar{U}^0_{\epsilon_k} = \pi^{l(\epsilon_{k+1})}_{l(\epsilon_k)} \left( \bar{U}^{N_{\epsilon_{k+1}}}_{\epsilon_{k+1}} \right)$;
    perform gradient flow at scale $\epsilon_k$;
  $\}$
$\}$

gradient flow at scale $\epsilon_k$ $\{$
  compute new image scales $\bar{T}_\epsilon = \mathrm{MGM}_{l(\epsilon_k)}(\epsilon_k)\bar{T}$, $\bar{R}_\epsilon = \mathrm{MGM}_{l(\epsilon_k)}(\epsilon_k)\bar{R}$;
  for each $n$ $\{$
    evaluate the gradient $\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}] = (\bar{T}_{\epsilon_k} \circledcirc (\mathbb{1} + \bar{U}^n_{\epsilon_k}) \ominus \bar{R}_{\epsilon_k}) \odot \left( h_{l(\epsilon)} \nabla_{h_{l(\epsilon_k)}} \bar{T}_{\epsilon_k} \circledcirc (\mathbb{1} + \bar{U}^n_{\epsilon_k}) \right)$;
    perform smoothing multi-grid V-cycle $\mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]$;
    maximize $\tau^n_{\epsilon_k}$ by Armijo's rule (Eq. 2.85);
    update solution $\bar{U}^{n+1}_{\epsilon_k} = \bar{U}^n_{\epsilon_k} \ominus \frac{\tau^n_{\epsilon_k}}{h_{l(\epsilon_k)}} \odot \mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]$;
    break loop if $\| \frac{\tau^n_{\epsilon_k}}{h_{l(\epsilon_k)}} \odot \mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}] \|_2^2 < \delta$;
  $\}$
$\}$
```

Algorithm 4.4 outlines the implementation of the image registration scheme (Eq. 4.9 on page 143) in pseudo-code notation. Each line corresponds to the configuration of the fragment processor with the corresponding kernels and the streaming of the texture operands through the so configured graphics pipeline. Some operations require several passes with slightly different configurations. The involved kernels are listed below.

- Smoothing with the multi-grid V-cycle $\mathrm{MGM}_{l(\epsilon_k)}(\sigma)$:
  - Restriction operator ρ^l_{l+1}.
  - Prolongation operator π^{l+1}_l.
  - Jacobi iterations with $A_{h_{l(\epsilon_k)}}$.
  - Residuum computation $\bar{U}^n_{\epsilon_k} - A_{h_{l(\epsilon_k)}}\bar{X}_{\epsilon_k}$.
- Energy functional:
  - Error computation $\bar{T}_{\epsilon_k} \circ (\mathbb{1} + \bar{U}^n_{\epsilon_k}) - \bar{R}_{\epsilon_k}$.
  - Evalutation of the energy gradient $\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]$.
- Utilities:
  - Multiply and accumulate $\bar{A} \cdot \bar{B} + \bar{C}$.
  - Bilinear interpolation $\sum_{i_x,i_y\in\{0,1\}} |i_x - \mu_x| \cdot |i_y - \mu_y| \cdot \bar{A}_{\alpha+(1-i_x,1-i_y)}$.
  - Evaluation of the lumped L^2 scalar product $\langle .,. \rangle_h$.

These kernels are executed by the Fragment Processor (FP), as an example Listing 4.1 on the next page shows the implementation of the Jacobi kernel in Cg. A different program in the Vertex Processor (VP) generates the texture coordinates in the structure `Frag2dIn IN` for the access to the neighboring nodes. The other parameters of `jacobiFP` are set in the application during the configuration process of the graphics pipeline (Figure 4.1 on page 112). Listing 4.2 on page 148 shows the pipeline configuration in a C++ program for the execution of one iteration of the Jacobi solver.

Because the FP accepts long configurations the kernels perform their task in one pass except for the lumped discrete L^2 scalar product. This can be evaluated with the `glHistorgram` function (Eq. 4.1 on page 125), but we discontinued the use of this function already for the diffusion processes because it is very slow. An alternative is to evaluate the scalar product by a component-wise multiplication and an iterative addition of local *texels*. Such a procedure involves a global access to all texels of a texture and would need a global register for accumulation, which is not supported in DX9 graphics hardware. Hence, we consider a hierarchical implementation with several passes. After the component-wise multiplication we consecutively halve the size of the resulting texture by applying local filters which sum up the local texel values. This step is repeated from the finest up to the coarsest grid level such that the final result of this hierarchical summation can be retrieved from the coarsest level as a single value for further processing by the CPU.

The energy computation $E_{\epsilon_k}\left[\bar{U}^n_{\epsilon_k} - \frac{\tau}{h_{l(\epsilon_k)}}\mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]\right]$ required in the evaluation of Armijo's rule (Eq. 2.85 on page 62) requires such a lumped discrete L^2 scalar product. Thus we compute

$$\begin{aligned} \bar{V}_\tau &:= \bar{U}^n_{\epsilon_k} - \frac{\tau}{h_{l(\epsilon_k)}}\mathrm{MGM}_{l(\epsilon_k)}(\sigma)\bar{E}'_{\epsilon_k}[\bar{U}^n_{\epsilon_k}]\,, \\ E_{\epsilon_k}\left[\bar{V}_\tau\right] &= \frac{1}{2}\langle \bar{V}_\tau, \bar{V}_\tau \rangle_{h_{l(\epsilon_k)}}\,, \end{aligned}$$

where the scalar product is implemented as the hierarchical summation described above.

Listing 4.1 Implementation of the kernel of the Jacobi solver for $A_h X = B$ in the graphics language Cg. Bold keywords belong to the language specification, italic ones are predefined types and functions of a self-made library which facilitates the access to neighboring nodes in a texture. Lines 9,10 assign local variables to data elements of the input data streams (given by the textures `Tex_B,Tex_X`), and the following lines define the actual processing of the data elements with the computation of the convolution and the linear interpolation: `lerp(a,b,c):=(1-c)a+cb`.

```
 1 FragOut
 2 jacobiFP( Frag2dIn IN,
 3           uniform sampler2d Tex_B : texunit0,
 4           uniform sampler2d Tex_X : texunit1,
 5           uniform float scale)
 6 {
 7   FragOut OUT;
 8
 9   float2 tex_B= f2texRECT(Tex_B, IN.cCoord.xy);
10   Stencil3x3r2 tex_X; texStar8(tex_X, IN, Tex_X);
11
12   float2 LN= ( + tex_X.mp + tex_X.cp + tex_X.pp
13               + tex_X.mc +           + tex_X.pc
14               + tex_X.mm + tex_X.cm + tex_X.pm )*(1/8.);
15   OUT.col= lerp(LN, tex_B, scale);
16
17   return OUT;
18 }
```

4.1.5.2 Results

The implementation runs on a NVIDIA GeForceFX 5800 Ultra 500MHz(4x2) / 500MHz(128 bit DDR). For DX9 GPUs it is, however, not possible to estimate the raw processing power from the clock frequencies, number of parallel pipes and memory bandwidth alone. The number of arithmetic processing elements (PEs), texture mapping units (TMUs) and their flexibility is very important, and GPU manufacturers have put different focus on these parameters, making a simple comparison impossible. It should suffice to say that the FX5800 is a fairly weak DX9 architecture, both NVIDIA GeForceFX 5900 and ATI Radeon 9800 have twice the processing power on computationally intensive problems.

The examples are composed of three different data sets: low and high frequency distortions (Figures 4.13 on page 149, 4.14 on page 150), large rigid deformations (Figures 4.17 on page 152, 4.15 on page 151) and medical data sets (Figures 4.18 on page 153, 4.19 on page 154). The corresponding figures show six different tiles, which are are arranged in the

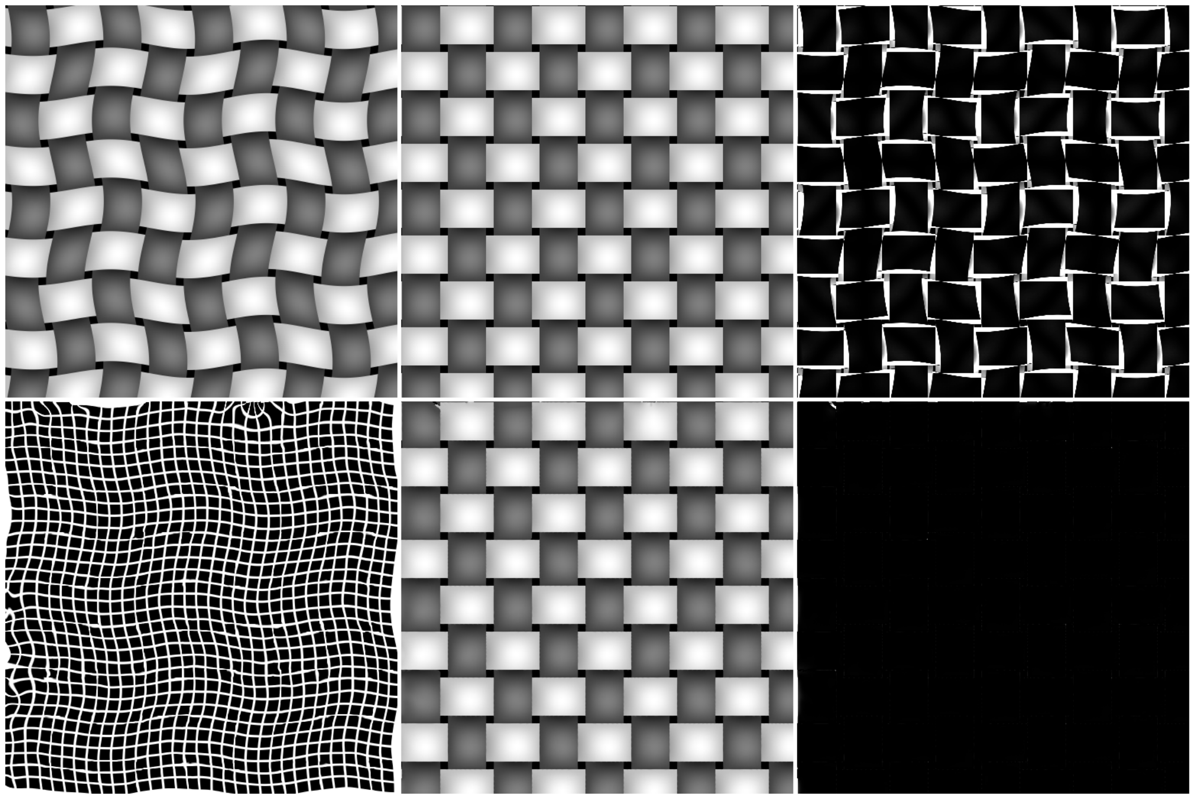
Listing 4.2 Configuration of the graphics pipeline and data streams for the execution of the Jacobi kernel in Listing 4.1. Bold keywords are functions of the Cg API, italic once are predefined arrays pointing to texture objects, vertex and fragment programs and their variables. The first line sets the target pbuffer for the output data stream. At the end (line 16) we release the pbuffer, such that it can be used as a texture operand in the next pass. Lines 6,7 configure the VP and FP with the kernel programs. Line 5 sets the `scale` parameter, lines 9,10 bind the textures `TEX_B`,`TEX_X` as input data streams for `jacobiFP` (Listing 4.1). Finally, line 12 sends the geometry of the current multi-grid level (`MGlev`) to the VP and thus initiates the execution of the Jacobi iteration in the graphics pipeline (Figure 4.1 on page 112).

```
 1 tex[TEX_N].toTexture(MGlev);
 2
 3 cgGLSetStateMatrixParameter(verVar[VP_NEIGH2D][VV_MVP],
 4   CG_GL_MODELVIEW_PROJECTION_MATRIX, CG_GL_MATRIX_IDENTITY);
 5 cgGLSetParameter4fv(fragVar[FP_JACOBI][FV_SCALE], scale);
 6 cgGLBindProgram(verProg[VP_NEIGH2D]);
 7 cgGLBindProgram(fragProg[FP_JACOBI]);
 8
 9 tex[TEX_B].bind(MGlev, GL_TEXTURE0_ARB);
10 tex[TEX_X].bind(MGlev, GL_TEXTURE1_ARB);
11
12 drawTex(tex[TEX_N].pos[MGlev], tex[TEX_N].size[MGlev],
13         tex[TEX_B].pos[MGlev], tex[TEX_B].size[MGlev],
14         tex[TEX_X].pos[MGlev], tex[TEX_X].size[MGlev]);
15
16 tex[TEX_N].fromTexture(MGlev);
```

following way: on the upper left we see the template $\bar{T}$ which should be deformed to fit the reference image $\bar{R}$ to the right of it; on the lower left we see the computed deformation $\bar{U}$ applied to a uniform grid and to the right the registration result, i.e. the template after the deformation $\bar{T} \circ (\mathbb{1} + \bar{U})$. The rightmost column shows the quadratic difference between the template and the reference image before $\|\bar{T}_\alpha - \bar{R}_\alpha\|_2^2$ (upper row) and after $\| \left(\bar{T} \circ (\mathbb{1} + \bar{U})\right)_\alpha - \bar{R}_\alpha\|_2^2$ (lower row) the registration. With one exception (Figure 4.19 on page 154) the differences are scaled with a factor 10 and grey values are clamped at black and white, otherwise one would hardly see anything on the error images after registration.

Figures 4.16 on page 152 and 4.21 on page 155 show the decrease of the energy against the overall number of update iterations in the process of registering the examples 4.15 on page 151 and 4.19 on page 154 respectively. Each point in the graph stands for a gradient descent step (first line in Eq. 4.9 on page 143), which includes the computation of the energy gradient, the smoothing with the multi-grid V-cycle, evaluation of Armijo's rule and the update of the solution (cf. Algorithm 4.4 on page 145). The graph discontinuities indicate scale changes

Figure 4.13 Elimination of low frequency distortions on 513^2 images in 9.8sec. We see that apart from the image boundary where sufficient information is missing, the deformation can be completely eliminated.

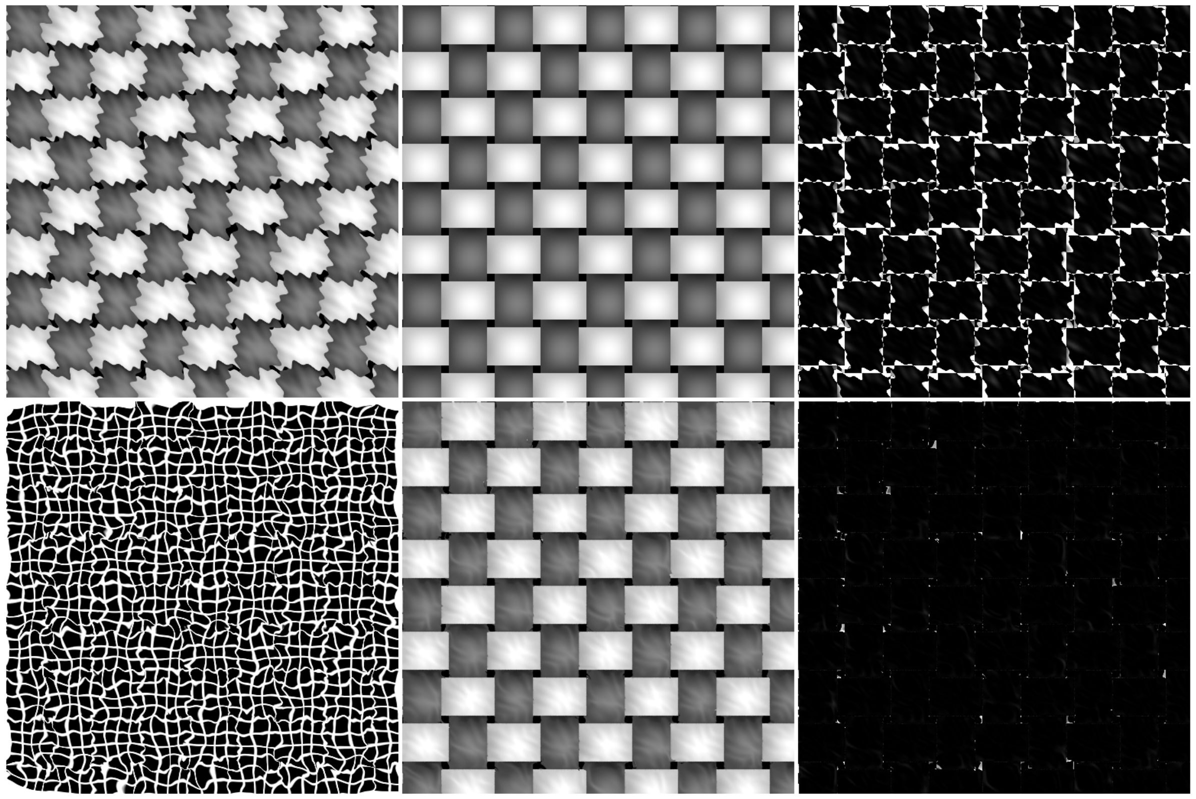


($\epsilon_k \rightarrow \epsilon_{k-1}$), while the X's on the x-axis represent changes of the grid level in the multi-grid hierarchy ($\Omega_{h_l} \rightarrow \Omega_{h_{l-1}}$). The number of the X's depends on the choice of the grid used for the computation of the initial deformation (Eq. 2.79 on page 60).

Usually each scale change increases the energy, because less smooth data is used in the computation of the local error. This effect can be sometimes particularly large for the last scale change, because on scale $\epsilon_0 = 0$ no smoothing of the images takes place. Introducing more scales, especially for the finest grid can lessen this effect, such that the energy graph looks nicer, but since the overall energy is hardly reduced in this way, we have not included these additional costly iterations on the finest grid in the standard parameter set. Sometimes we also observe energy decreases at the time we change the grid level in the multi-grid hierarchy (Figure 4.21 on page 155). This effect is due to the additional smoothing caused by the prolongation operator, which decreases the local error in areas of smooth deformations.

The computations were performed either in the s23e8 full float or the graphics specific s10e5 half float format (cf. Table 4.3 on page 124). The results for both formats are very similar which is an indicator for the stability of the algorithm. The performance varies only slightly in favor of the smaller format as there exists a sufficient bandwidth in comparison to the number

Figure 4.14 Elimination of high frequency distortions on 513^2 images in 9.5sec. The algorithm performs well for high frequencies. Only some ambiguous situations around the strongly deformed small dark squares can not be resolved.

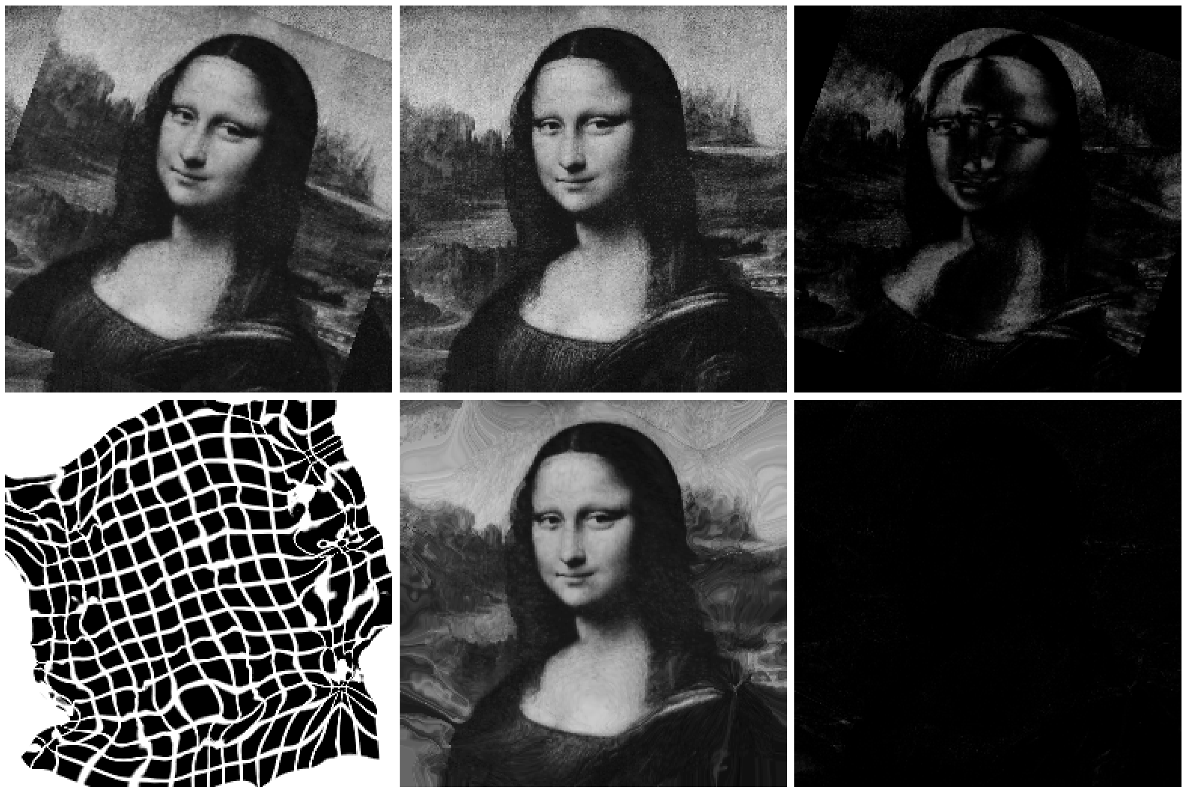


of operations in the kernels to transport the full float values. The standard parameter set uses 15 scales, up to 10 iterations of the update loop and up to 10 iterations of the energy loop. We say 'up to' because the loops are aborted if the update is too small. The smoothing multi-grid V-cycle uses 3 Jacobi iterations on each grid both up and down the V-cycle.

The duration of the registration depends on the size of the images and the number of actual passes, since several loops allow adaptive loop abortion. In general, the registration of 257^2 images takes approx. 3sec and up to 10sec are needed for fully distorted 513^2 images. But for even larger (513×769) medical data, often less time is required (8.5sec) because for such data the deformations are usually not so severe (Figure 4.19 on page 154). An estimated software performance for this data set based on the highly optimized implementation in [Clarenz et al., 2002], which actually deals with 3D data, would amount to approx. 34sec and thus 4 times more time than the graphics implementation requires. As explained in the beginning of this section, this factor would be twice as large for the current DX9 GPUs and presumably four times for DX9+ GPUs.

Beside the increasing hardware performance, we also expect efficient improvements in the APIs. The render-to-texture mechanism with *pbuffers* (Section 4.1.2.1 on page 119) made

Figure 4.15 Registration of a rotated 257^2 image in 5.9sec. This is a very hard test for the non-rigid registration, which takes exceptionally long to finish for this image size, since without any a priori knowledge about the underlying rotation there are many possibilities to match the similar grey levels against each other. Obviously in the area of the body the inverse rotation could be identified by the algorithm, whereas the background is rather poorly registered. This outlines that even without the guidance of a concrete deformation model the method performs well for large deformation if the structures to be matched are sufficiently pronounced.



unnecessary data movement obsolete, but the switches between pbuffers are still fairly heavy weight. Since these switches occur extremely often, after each operation, the developping mechanism for light weight pbuffer switches will optimize the global data-flow on GPUs.

Figure 4.16 Energy decrease in registering the images from Figure 4.15 on the page before. The structure of the graph is explained on page 148.

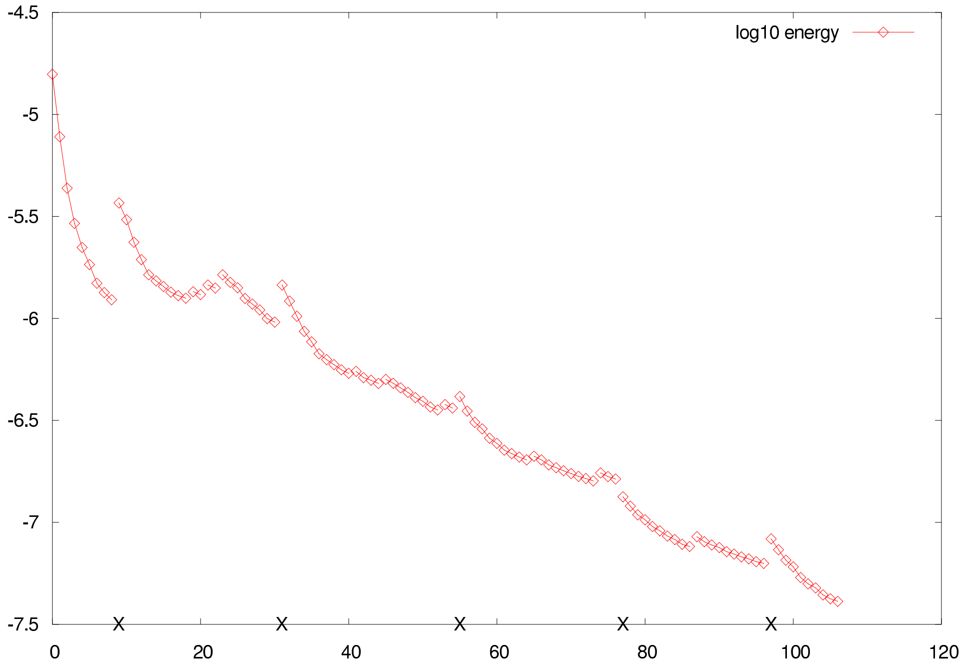


Figure 4.17 Registration of a large scale rigid deformation on 257^2 images in 3sec. The multi-scale approach allows to reconstruct even large deformations. However, without assumptions on the deformation model, the computed deformation might differ from our expectations.

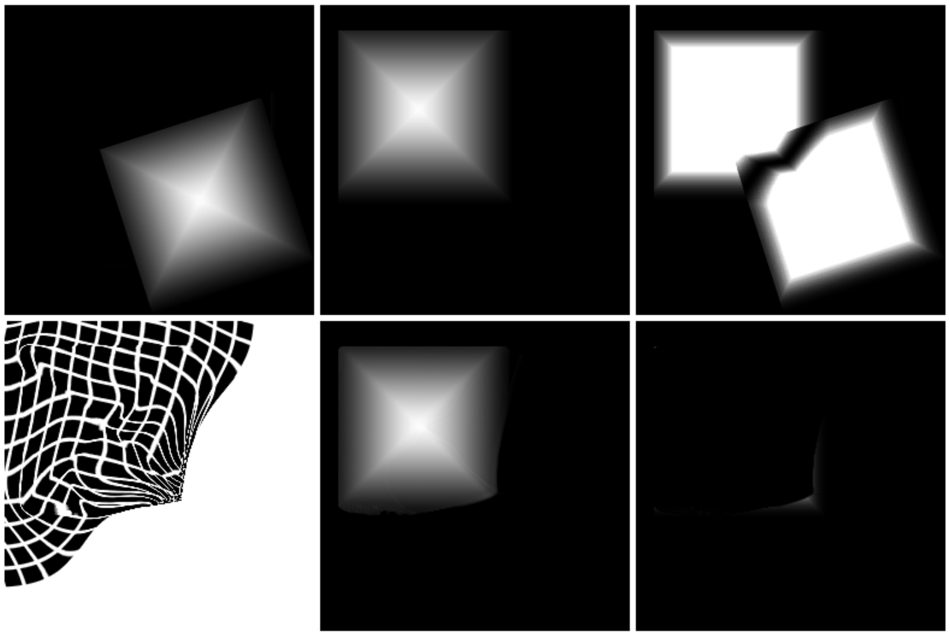


Figure 4.18 Elimination of a possible acquisition artefact for a medical 257^2 image in 2.2sec. Here we have an example were some areas must expand while others must shrink to fit the reference image. The matching works well apart from the small fluctuation in the lower left part. Such deviations are striking to our perception but have little effect on the overall energy, because they reflect mismatches of the morphology rather than grey level deviations. Therefore additional morphological energy components will be considered in the future.

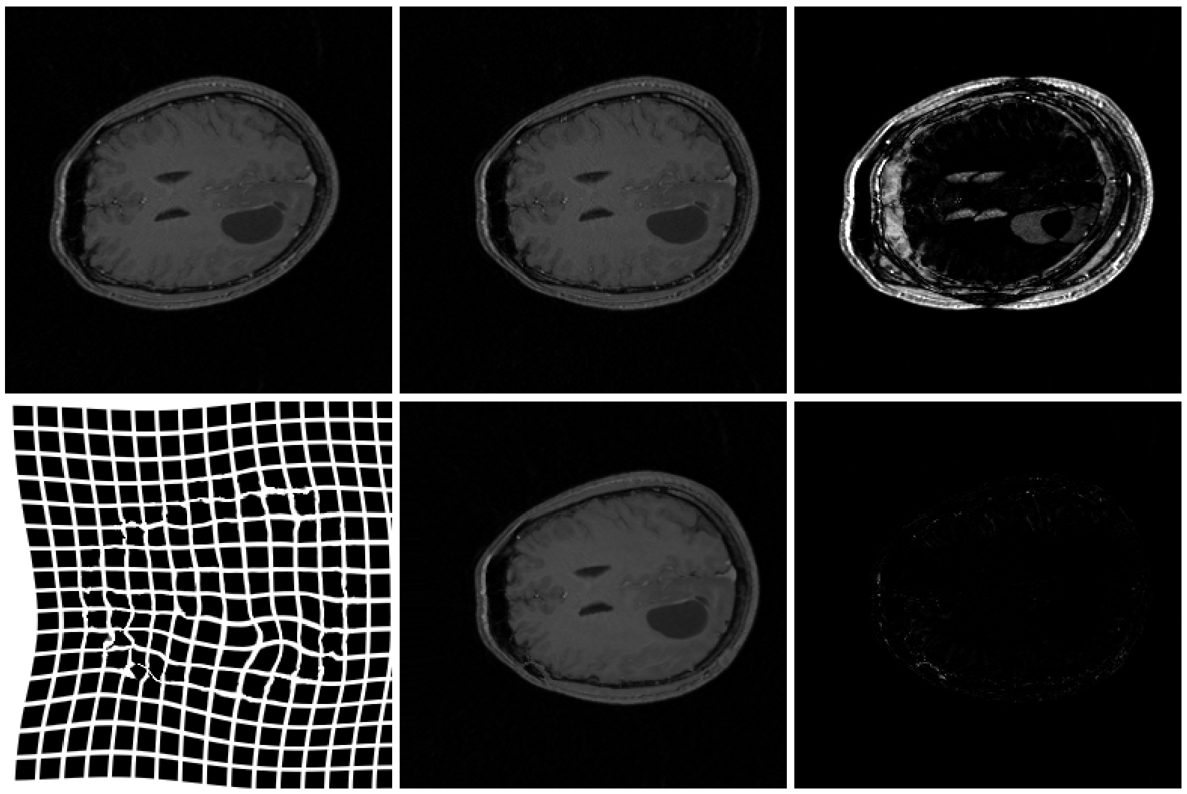


Figure 4.19 Registration of two brain slices of the same patient taken at different times on 513×769 images in 8.5sec. As the slices were extracted from 3D volumes some anatomical structures are not present in both images and lead necessarily to an error in the registration result, especially in the upper right parts of the images. Also in contrast to the other examples the grey values of the corresponding structures have not exactly the same value such that a small error is present even in the case of perfect fit of image edges (left scull). For this reason, here the error images on the right are not scaled.
The algorithm, however, is not distracted by the different grey levels and complements missing data fairly smoothly. In particular the edges are matched nicely. Figure 4.20 on the facing page shows the scaled error in the interior of the images.

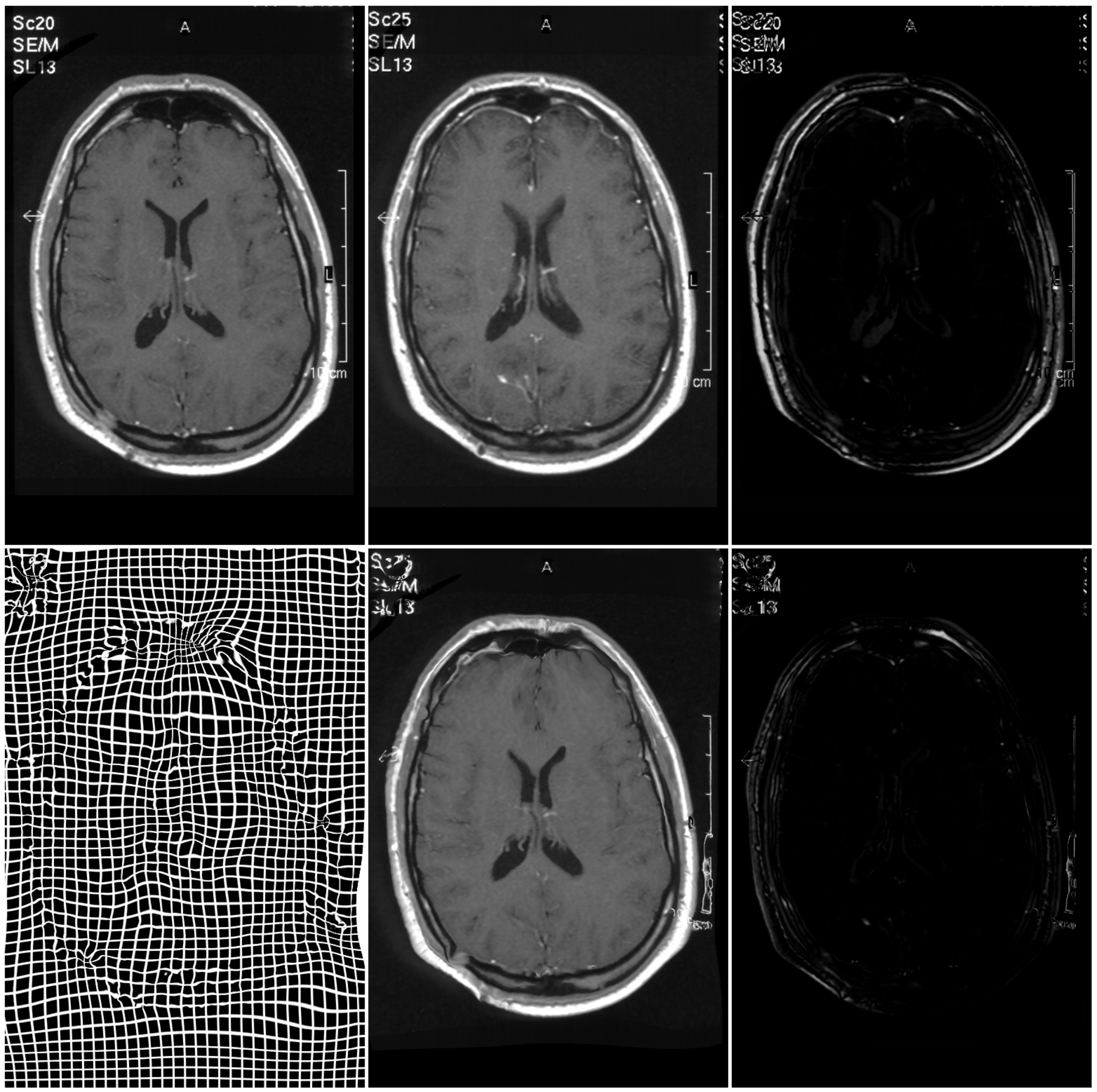


Figure 4.20 The enlarged central part of the error images from Figure 4.19 on the preceding page. Here the error has been multiplied by 10 again. The images demonstrate that the algorithm does also a good job in matching the central areas.

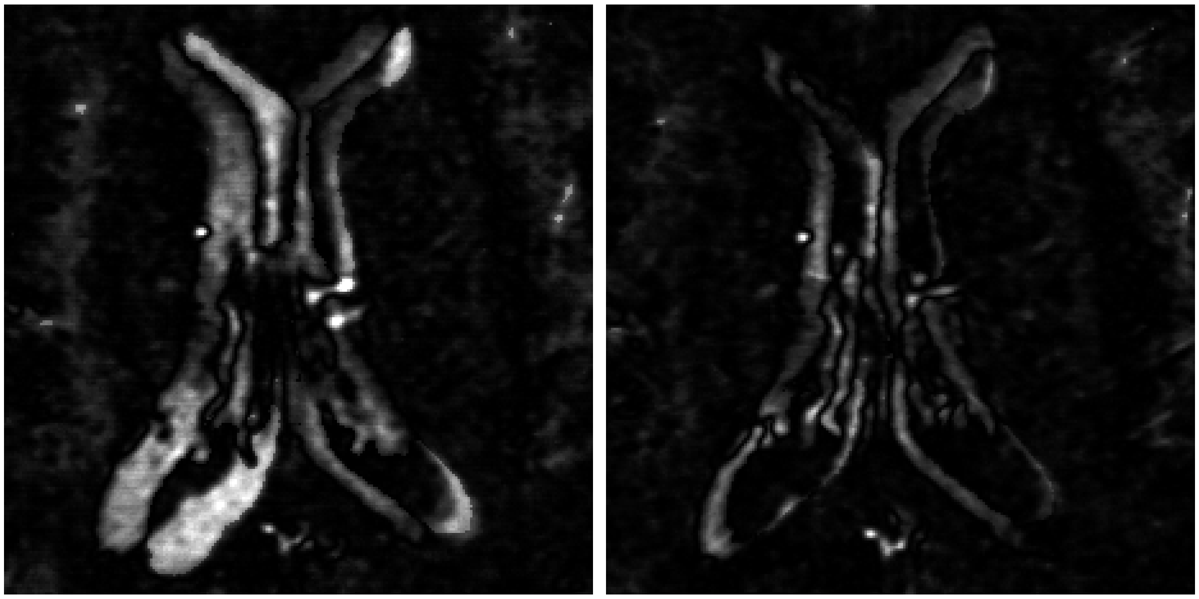
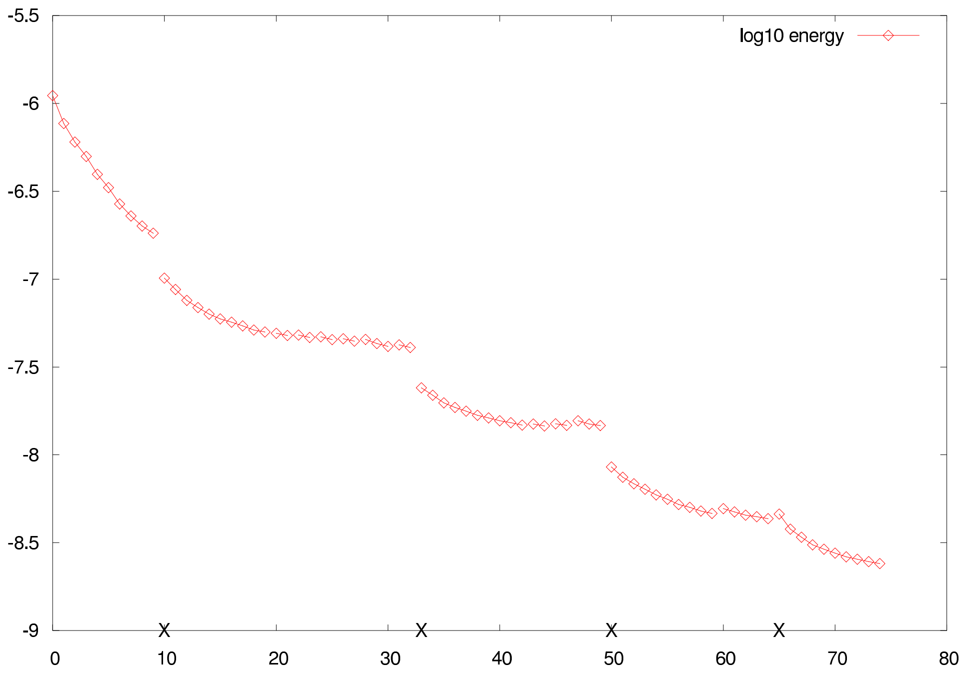


Figure 4.21 Energy decrease in registering the images from Figure 4.19 on the facing page. The structure of the graph is explained on page 148.



4.2 Reconfigurable Logic

Field Programmable Gate Arrays (FPGAs) evolved since the mid 1980s from small rapid-prototyping devices to multi-million gate chips with large blocks of integrated memory, multipliers and special circuits or even embedded processors. Sections 3.2.4 on page 93, 3.3.1.4 on page 100 introduce into the subject, while [Bondalapati and Prasanna, 2002; Brown and Rose, 1996; Compton and Hauck, 2002] explain also other RL architectures and present an overview of many systems. Our implementation does not use a very sophisticated device, but the main considerations about efficient *spatial computing* come here into play just the same [DeHon, 2002]. The work has been done in collaboration with Steffen Klupsch and Markus Ernst from the institute for Integrated Circuits and Systems Lab of the Technical University Darmstadt headed by Prof. Sorin Huss, it appeared in [Klupsch et al., 2002]. This section is based on the publication and associated material from presentations. First we present the hardware platform and the design flow, and then the concrete implementation.

4.2.1 FPGA Card

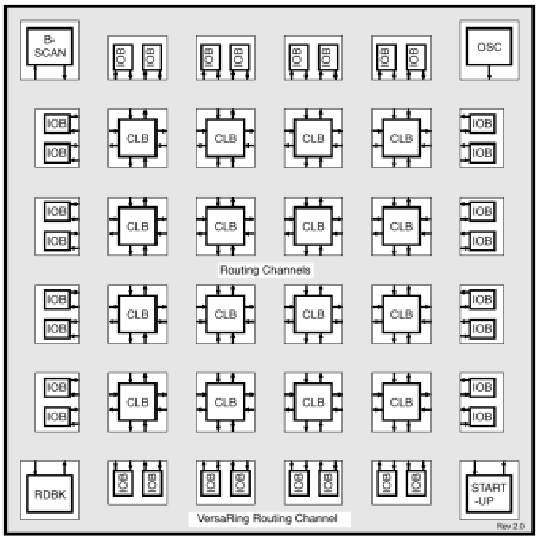
Here we describe the functionality and programming model for the simple FPGA card we have used. In principle the same aspects apply to FPGA systems in general, but often FPGA cards contain more components, both multi-FPGA arrangements and hybrid hardware-morphware combinations, so that the interaction patterns are more complicated.

4.2.1.1 Technology

We use a PCI accelerator card [Silicon Software, 1999], equipped with a XC4085XLA FPGA from Xilinx [Xilinx Inc.] for the implementation. The chip contains a 56x56 Configurable Logic Block (CLB) array (Figure 4.22 on the next page on the left). Each CLB contains several lookup tables (LUTs) which are configured to perform logic operations on the input lines (Figure 4.22 on the facing page on the right). The overall capacity of the XC4085XLA equivalents approx. 180k system gates, i.e. primitive logic operations in an ASIC. This is very little compared to mulit-million gate chips nowadays, which also contain embedded hardwired special circuits and a lot of memory. In particular the lack of dedicated local RAM in the chip, means that many LUTs must be consumed to define local memory rather than serve as computational resources.

The card has a programmable clock generator and 2MiB SRAM accessible directly from the FPGA. A schematic overview of the card with the bandwidth of the different buses is shown in Figure 4.23 on the next page on the left. The card itself has no sophisticated host processor and needs external control for the configuration process. We connected it via the PCI interface to a PC. In this respect the technology is very similar to graphics cards which also use a host for the general control functionality.

Figure 4.22 Layout of a FPGA architecture (Xilinx [Xilinx Inc.]). On the left we see the general arrangement of the Configurable Logic Blocks (CLBs) in an array. Between the CLBs there are also configurable routing channels. On the right the layout of a CLB with lookup tables (LUTs) as main computational resources is displayed.



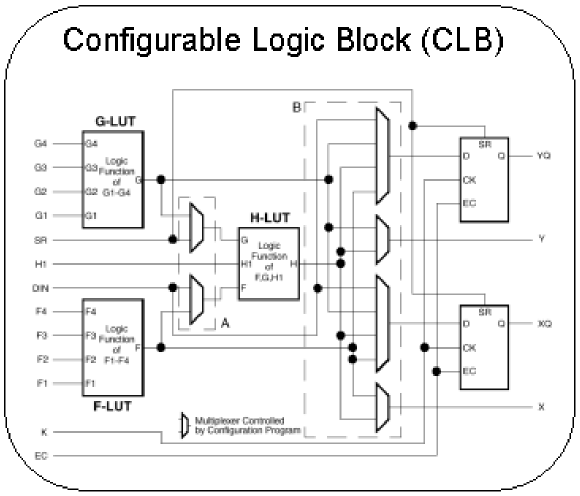
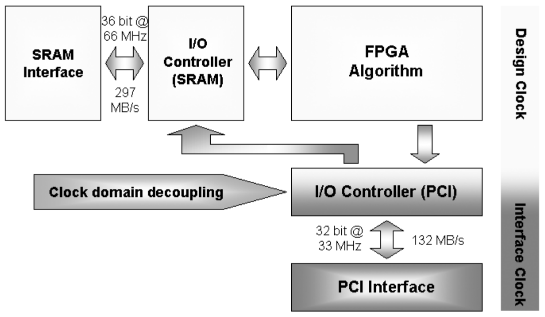
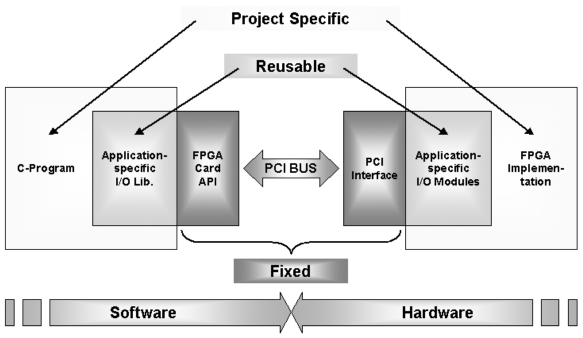


Figure 4.23 A schematic view of our FPGA card and its system partitioning. The 66MHz access to the SRAM is a maximal value and can only be achieved with hand optimized pipelined access, but then the full bandwidth is available. The peak bandwidth of the PCI bus on the other hand cannot be reached as the bus must share resources with other components in the PC.





4.2.1.2 Programming

A generic concept for the system partitioning based on the PCI conncection is shown in Figure 4.23 on the preceding page on the right. The connection is centered around the fixed PCI interface with reusable, but application specific, low level IO access functions. The algorithmic functionality is divided into a software part running on the CPU, and a *flowware* and *configware* (Table 3.2 on page 85) part configured in the FPGA. The software part holds the general control of the process and has been coded in C. The design of the sources for the execution on the FPGA is more sophisticated.

Nowadays, the design entry for most digital circuits is based on behavioral models in Hardware Description Languages (HDLs) such as Verilog, or VHDL which we used. These descriptions are processed by synthesis tools to derive a netlist of basic logic elements, which can be fed into place&route tools. The result is a bitstream used for the configuration of the FPGA. Unlike the changing of code in a High Level Language (HLL) and a subsequent quick re-compilation to a micro-processor's assembly language, the transition from a changed Hardware Description Languages (HDLs) model to the bitstream is not an interactive process. It takes tens of minutes to hours for even relatively small projects like ours. For large FPGA these times can easily grow to days, depending, of course, on the architecture on which the tools run. There is also a trade-off between the execution time and the target requirements (mainly clock cycle) of for the synthesis process.

Because the tools use heuristics to fullfil the requirements a little change in the model or slightly different target clock cycle can result in a completely different bitstream after the change. This makes debugging very hard, since errors may produce different effects depending on the actual bitstream. To avoid the place&route step after each change, there also exist simulators for the models. But clearly the clock accurate similiuation takes its time and some errors are so delicate that they only show up in the on-chip execution and not in the simulator. Recapitulating we can say that despite many tools the configuration design for fine grain architectures is still very demanding and an art of its own.

The main option to accelerate the design and reduce the error-prone redevelopment of logic circuits is the incorporation of higher abstraction levels into synthesis tools, which we also exploited for our design. By using parametrised high level descriptions, we gain the possibility to do detailed design space explorations. The design complexity is reduced by using small abstract behavioral descriptions. By extensive hardware/software co-evaluation we add validation of the synthesis results without leaving the abstraction level [Klupsch, 2001]. The time saved is then used for selective optimization of critical design subcircuits.

We should mention that there exist compilers which translate C or certain dialects of it to a HDL model [Frigo et al., 2001; Hammes et al., 1999]. The idea is to further increase the abstraction level and this is particularly important for large designs. But these approaches must necessarily omit some of the low level optimization options. Therefore, a two stage approach is sometimes take, where the overall design is done in C but individual parts are modelled in a HDL. This is similar to the inclusion of assembler code in HLL for a micro-processor. In our implementation we have not made use of this strategy, because the small FPGA forced us rather in the opposite direction, coding the overall design in HDL and hand optimizing the netlists and their placement for critical parts of the configuration.

4.2.2 Computations

The low level reconfigurability allows very high flexibility. Both in the design of an efficient data-flow and computations the available resources can be used economically with the exact amount needed.

4.2.2.1 Data-Flow

The global data on the cards is as follows. The software part uses the PCI bus to configure the FPGA, load the input images to the SRAM of the card and retrieve the results after execution. The flowware on the FPGA consist of the SRAM controller which generates and stores data streams from the SRAM processed by the computational units. The configware defines the configuration of these units.

Although we use a fairly small FPGA, there is still more parallelism available than could be served by the SRAM bus. For problems with high *computational intensity* this is not crucial, but image processing problems are rather memory intensive. So the guiding idea is to efficiently utilize the available memory bandwidth by reading each pixel of an image only once, and caching it on the chip for reuse [Benitez, 2003; Hammes et al., 2001]. If a 3 by 3 neighborhood is needed in a formula for example, one has to cache two lines on the chip to avoid rereading of values from external memory. On newer FPGA dedicated RAM blocks embedded in the chip can be used for this purpose. In our case where no RAM blocks are available, we had to use some of the CLBs for local memory. By rule of thumb a CLB can be used for storing of 32 bit. So the entire chip (3136 CLBs) can hold 100kb, but then we would have no resources for the computations.

4.2.2.2 Operations

Since FPGA can be configured to execute any combination of boolean function it is clear that they can implement any functionality. After all, micro-processors and other ASICs are also implemented in this way and FPGAs are still used to simulate these designs. But for a given application it would be a waste of resources to implement a standard processor design and then solve the problem with this configured processor. The advantage of the low level programmability of FPGA is namely the possibility to adapt the problem solver exactly to the needs of the application. This includes the use of non-standard number formats, application specific operations and pipelines.

Table 4.4 Area consumption and latency of some operations in Reconfigurable Logic. The variable B_a refers to the bitlength of the arguments, B_c to the bitlength of the results.

operation	area	latency
$c_0 = \min/\max(a_0, a_1)$	$B_a + 1$	2
$c_0 = a_0 \pm a_1$	$2B_a$	B_a
$c_0 \pm c_1 = a_0 \pm a_1 \pm a_2$	$2B_a$	1
$c_0 = a_0 \cdot a_1$	$\approx B_a(B_a - 1)$	$\approx B_a \log_2 B_a$
$c_0 = \sqrt{a_0}$	$\approx 2B_c(B_c - 5)$	$\approx B_c(B_c + 3)$

An application specific number format can save a lot of resources. Table 4.4 lists the resources and latencies of several operations. So a 16 bit multiplier consumes almost twice (1.87) the number of resources as a 12 bit multiplier. But in *spatial computing* (Section 3.3.1.4 on page 100) resource gains translate directly into performance, since two parallel 12 bit multipliers could be implemented in almost the same area. Moreover, smaller number formats do not imply losing precision in intermediate computations, since we can use ***variable precision arithmetic***. This means that intermediate results are resolved with as many bits as necessary for the desired accuracy, e.g. in an addition $c = a + b$ of numbers resolved in B_a, B_b bits the result c should have $B_c = \max(B_a, B_b) + 1$ bits to retain full precision. In a multiplication $c = a \cdot b$ we would need $B_c = B_a + B_b$ for full precision, but we can represent c with less bits if this suffices for the envisioned accuracy in our application. In an iterative scheme it would be unreasonable to increase the format size with each iteration, but we can maintain an error free calculation throughout the pipeline and apply one of the *rounding modes* (Eq. 2.22 on page 32) only at the very end.

There are many options for application specific commands with configurability at the bit level. Something we usually do not encounter in software programs are redundant number representations. If many operations of the same kind have to take place, e.g. $\sum_{i=1}^{N} a_i$, then computing not uniquely defined c_i such that $\sum_{i=1}^{N} a_i = \sum_{i=1}^{N-1} c_i$ can be performed much faster than $c = a_N + a_{N-1}$ (cf. Table 4.4). So for the whole sum the speedup is enourmous.

Finally, options for parallelism both *in depth* and *in breadth* (Section 3.2.2 on page 87) are multitude on FPGAs. In *pipelining*, for example, one can introduce many pipeline stages for even small computing units, because the atomic units (LUTs) are very fine. As a rule of thumb, longer pipelines enable higher clock frequencies and thus more *throughput* (cf. Section 3.2.1 on page 84). In the example of Figure 3.4 on page 86 one could divide the adders and the multiplier in even faster clocking stages.

4.2.3 Level-Set Segmentation

We implemented a solver for the *level-set equation* used for image segmentation on the FPGA card described in Section 4.2.1.1 on page 156. The implemented model is exactly the same as in Section 4.1.3 on page 127, where a graphics hardware implementation has been presented. In particular, we encorporate only external forces for the movement of the interface.

Because image processing algorithms are so well suited for parallization and FPGAs have become a mass market, literature on implementations in RL is extensive. For a comprehensive overview of FPGAs based DSP applications including image and video processing we refer to [Tessier and Burleson, 2001]. Related work on image segementation can be found in [Demigny et al., 2000; Tian et al., 2003; Waldemark et al., 2000].

4.2.3.1 Implementation

The quantized upwind scheme for the *level-set equation* (Eq. 2.73 on page 57) reads:

$$\begin{aligned}
\bar\Phi^{n+1}_\alpha &= \bar\Phi^n_\alpha \ominus \bar g^{\mathrm{lin}}_\alpha(D^-\bar\Phi^n, D^+\bar\Phi^n) \\
g^{\mathrm{lin}}_\alpha(U,V) &:= (\tfrac{\tau^n}{h}\bar F)^+_\alpha \odot \|(\bar U^+_\alpha, \bar V^-_\alpha)\|_{\mathrm{lin}} \oplus (\tfrac{\tau^n}{h}\bar F)^-_\alpha \odot \|(\bar U^-_\alpha, \bar V^+_\alpha)\|_{\mathrm{lin}} \\
\|X\|_{\mathrm{lin}} &:= c \odot \|X\|_1 \oplus (1-c) \odot \|X\|_\infty \\
D^+_\alpha\bar\Phi^n &:= \begin{pmatrix} \bar\Phi^n_{\alpha+(0,1)} \ominus \bar\Phi^n_\alpha \\ \bar\Phi^n_{\alpha+(1,0)} \ominus \bar\Phi^n_\alpha \end{pmatrix} \\
D^-_\alpha\bar\Phi^n &:= \begin{pmatrix} \bar\Phi^n_\alpha \ominus \bar\Phi^n_{\alpha-(0,1)} \\ \bar\Phi^n_\alpha \ominus \bar\Phi^n_{\alpha-(1,0)} \end{pmatrix},
\end{aligned} \tag{4.10}$$

with the quantized operations $\{\oplus, \ominus, \odot\}$ corresponding to their natural analogs $\{+, -, \cdot\}$ (Section 2.2.3.1 on page 36). The speed function $\bar F$ is the discretization of

$$f(x) = c(x) + g_1(p(x)) + g_2(\|\nabla p(x)\|)\,, \tag{4.11}$$

which depends on the image intensities $p(x)$ and its gradient modulus $\|\nabla p(x)\|$. The functions c, g_1, g_2 and their parameters are set by the user (cf. Section 2.1.2.1 on page 19). The initial level-set $\bar\Phi^0$ function is interactively generated by the user by specifying some points in the image as the starting points of the level-set evolution (cf. Figure 4.4 on page 133).

In case of graphics hardware (Section 4.1.3.1 on page 128) we used the approximation of the Euclidean norm, because there was no way of accurately evaluating this two dimensional non-linear function in the GPU at that time. As discussed in Section 4.2.2.2 on page 159, in a FPGA any function up to a given precision can be evaluated. Here, we apply the approximation because of the comparably high resource demand of the Euclidean norm (cf. Table 4.4 on the preceding page). The multiplications in the approximation on the other hand are very cheap if we choose $c = \frac{a}{2^n}$ with a small nominator a. Since we deal with an approximation anyway, c does not have to equal exactly one of the suggested choices (cf. Eq. 2.72 on page 56).

Figure 4.24 The caching strategy for the level-set solver in Reconfigurable Logic. On the left we see the required input data for an update unit which evaluates the scheme from Eq. 4.10 on the page before and how caching of 2 image lines reduces the amount of input pixels to two per clock cycle. On the right two update units work in parallel, which leaves the size of the caches equal, but doubles the required input pixels to 4 and output pixels to 2.

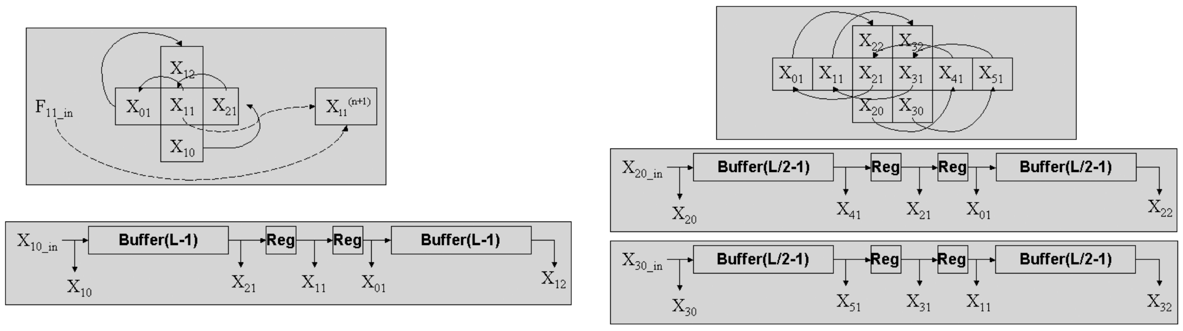
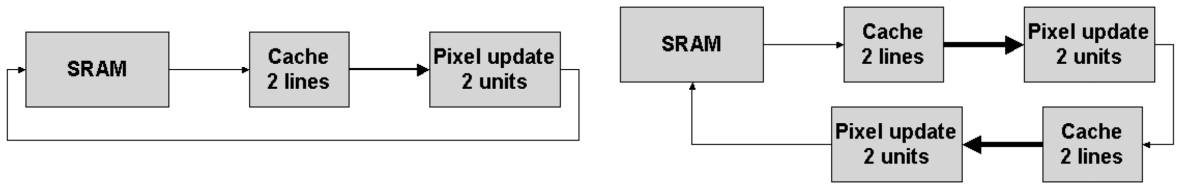

F_{11_in}
X_{12}
X_{01} X_{11} X_{21}
X_{10}
$X_{11}^{(n+1)}$
X_{10_in}
Buffer(L-1)
Reg
Reg
Buffer(L-1)
X_{10} X_{21} X_{11} X_{01} X_{12}
X_{22} X_{32}
X_{01} X_{11} X_{21} X_{31} X_{41} X_{51}
X_{20} X_{30}
X_{20_in}
Buffer(L/2-1)
Reg
Reg
Buffer(L/2-1)
X_{20} X_{41} X_{21} X_{01} X_{22}
X_{30_in}
X_{30} X_{51} X_{31} X_{11} X_{32}

Figure 4.25 The pipeline arrangement for the level-set solver in Reconfigurable Logic. On the left the number of update units which evaluates the scheme from Eq. 4.10 on the preceding page has been increased so far as to meet exactly the available bandwidth to the SRAM. More parallelism can only be reached in depth which requires the duplication of both the caches and the pixel update as can be seen on the right.



A good compromise between accuracy and performance was to use a 12 bit signed *fixed point number* format for the data representations. In intermediate calculations *variable precision arithmetic* ensures exact computation until the final result when $\bar{U}_\alpha^{n+1}$ is rounded and clamped to the 12 bit format again. Because of this format reduction the synthesis tool even eliminates some computations which influence bits which are irrelevant after the rounding. For this to happen the correct tie breaking strategy among the *nearest rounding modes* (Eq. 2.22 on page 32) must be chosen. The update unit, which executes the entire formula (Eq. 4.10 on the preceding page), was automatically divided into several pipeline stages to find a good compromise between area count and maximal attainable clock frequency.

In contrast to the DX7 GPUs (Section 4.1.3.1 on page 128), the FPGA offers so much parallelism, that we must now focus on minimal data movement (Section 4.2.2.1 on page 159) to ensure that several pixels can be processed in parallel. At each node α five $\bar{U}^n$ values and one $(\frac{\tau^n}{h}\bar{F})$ value is needed for the computation of $\bar{U}_\alpha^{n+1}$. If we cache two lines of the image than only one $\bar{U}^n$ and one $(\frac{\tau^n}{h}\bar{F})$ value are needed in each clock cycle. The other values can be retrieved from the internal caches (see Figure 4.24 on the preceding page on the left). Because the entire lines are cached, it suffices to read one more $\bar{U}^n$ and $(\frac{\tau^n}{h}\bar{F})$ value in the same clock cyle to be able to feed all necessary values to a second update unit (Figure 4.24 on the facing page on the right). This increases the IO demand to 4 pixels input and 2 output. But the 36 bit SRAM bus can transport only three 12 bit values per clock cycle. The solution is to run two different clock frequencies on the FPGA. The SRAM controller which performs the reads and writes to the external SRAM memory has a unit which operates at 66MHz, and thus can fulfill the IO demand of 6 pixels for the two update units running at 33MHz. The double clocked logic operates extremely close to the theoretic limit of the SRAM bus and had to be placed manually to ensure short paths which allow this high clock frequency.

More *parallelism in breadth* was not possible since it would further increase the IO demand, which already operated at its very limit. But after the implementation of the two update units and the cache we had a pipeline (Figure 4.25 on the preceding page on the left) which left many CLBs unused. In such cases when the *bandwidth* is restricted but computational resources are still available, *parallelism in depth* can be applied. In our case this meant duplicating the cache and update unit logic and using it for a second iteration (Figure 4.25 on the facing page on the right). This filled the FPGA to over 80% and it exhausted the routing capabilities so no futher parallelization was possible (Figure 4.26 on the next page). For a larger FPGA this process of appending dublicated logic could be continued much longer, which nicely demonstrates how the *spatial computing* approach directly gains performance from a larger spatial computing domain.

4.2.3.2 Results

Figures 4.27 and 4.28 on page 165 present example segmentations computed with the FPGA. With the two update units delivering an output pixel in each clock cycle in parallel at 33Mhz we have a throughput of 66Mpixel/s. Because we compute two iterations in one go this equivalents a throughput of 132Mpixel/s per iteration. Thus we can perform approximately 2000 iterations per second on a 256^2 image, or equivalently one iteration takes 0.5ms. This is eight times faster than the fragment blending based implementation in graphics hardware, although the throughput of the graphics pipeline for an individual operation was 1000 Mpixel/s (Section 4.1.3.2 on page 130).

The advantage of the FPGA pipeline is, that it does much more work, namely obtain two solution pixels, in each clock cycle. The GPU used only one operation per pass and moreover it had to read back the result before processing it again. On DX9 GPU one could also encode the whole update formula in a single configuration for the FP. But the FP is a hardwired unit, so that the advantage of the long pipeline could still not be realized. Instead, the several pipelines would attack the problem in breadth. But on modern FPGA there would be also much more space for parallelization, so that the performance crown would most likely be still with the Reconfigurable Logic (RL) implementation.

Figure 4.26 Final layout of the level-set solver in the FPGA. The term 'Pixel Engine' in the legend refers to the composition of 2 pixel update units and 2 cache lines for the $\bar{\Phi}$ values (cf. Figure 4.25 on page 162). We see that after the heuristic place&route step there is only little spatial coherency left in the functional units. On the small scale no correspondence between individual operations of the algorithm like additions and mulitplications and the CLBs can be found, because these operations may spread and mix arbitrarly across the lookup table functionality of several CLBs. Debugging at this level is almost impossible and must take place in the simulator or through the output of control values.

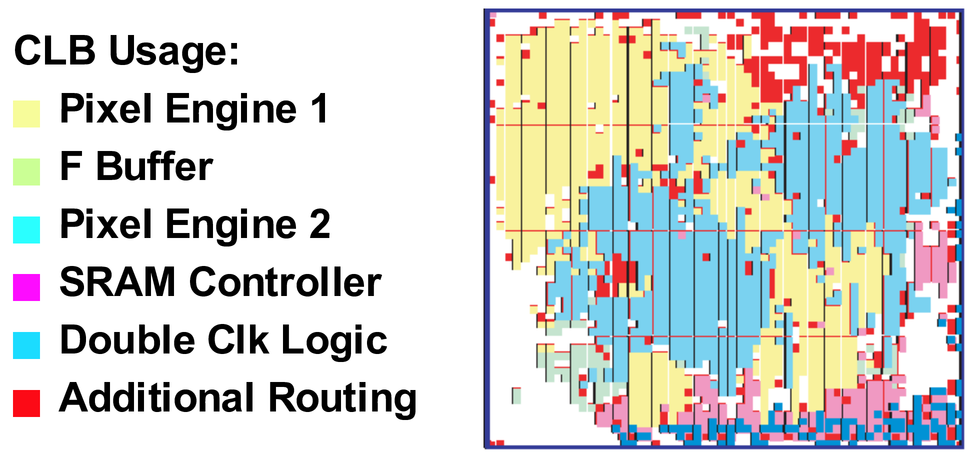


Figure 4.27 Segmentation result computed in a FPGA.

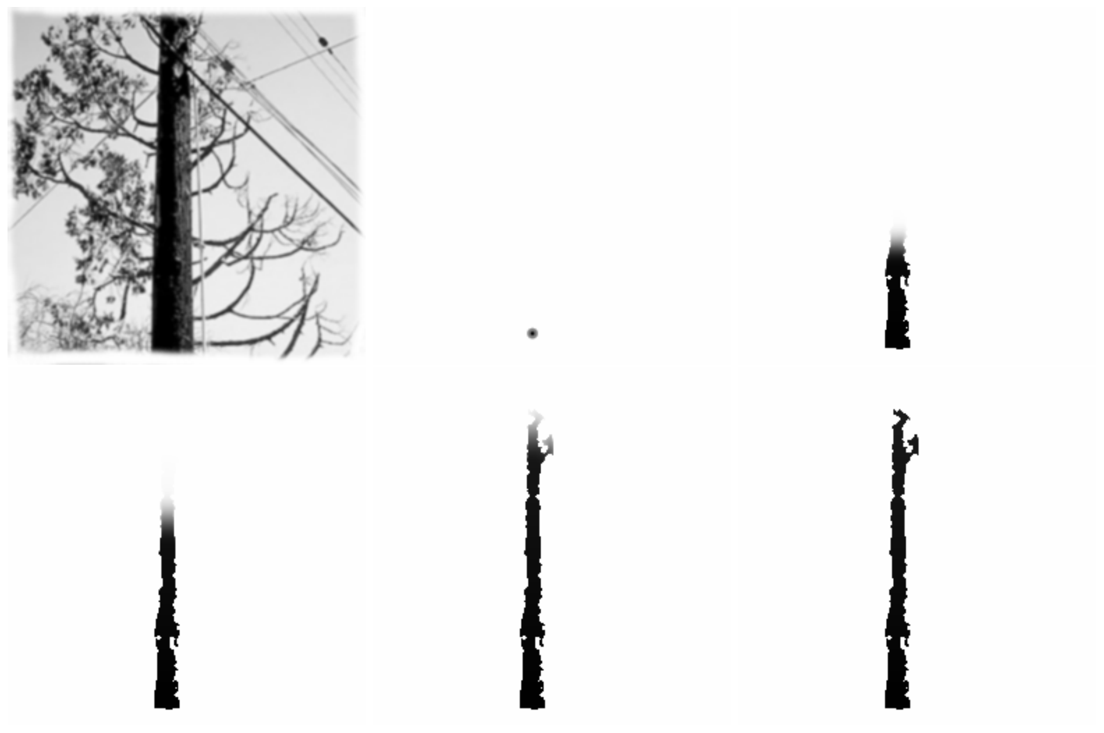
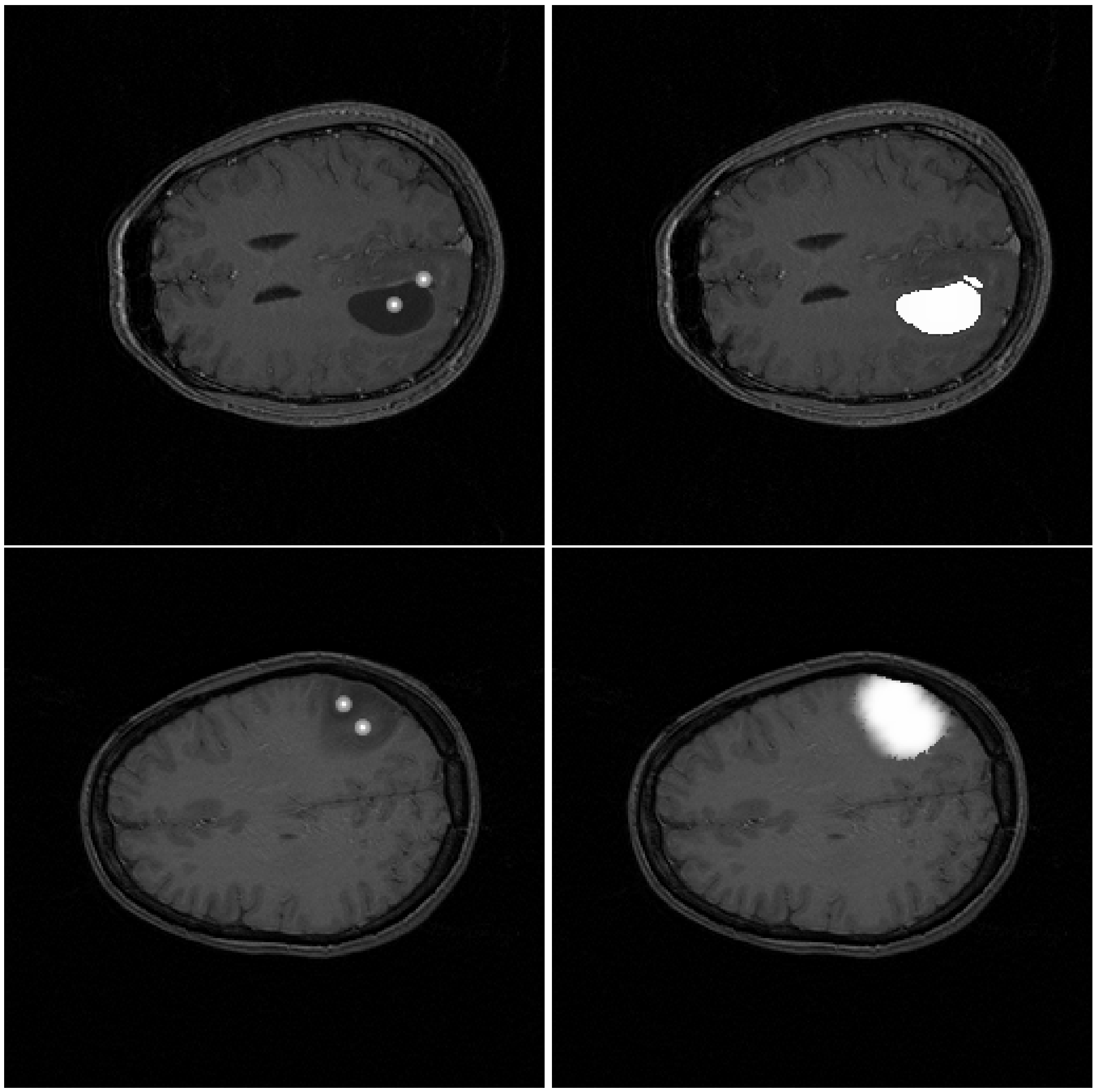


Figure 4.28 Segmentation of tumors computed in a FPGA.



4.3 Reconfigurable Computing

Reconfigurable Computing (RC) concepts and architectures have experienced a rapid evolution since the mid 1990s. While [Hartenstein, 2001] offers an excellent survey of this development, we refer back to Sections 3.2.4 on page 93, 3.3.1.5 on page 102 for a short introduction. But while GPUs and FPGAs have become a mass market, RC machines are still at the beginning of the exponential grow. Typical for this stage of development is the huge variety of competing designs with little standardization among them. Companies working in this field are often only several year old start-ups which evolved directly from research activities at universities, e.g. [Elixent; IPflex; PACT; picoChip; QuickSilver]. These companies basically do not produce chips themselves but offer to synthesize a chip design for fabrication, or licence IP-cores. To evaluate the chip's properties before fabrication one uses clock cycle exact chip simulators. We thank PACT [PACT] for providing a 3 months free research license for a simulator of their XPP architecture in 2002. In the following we describe the architecture and its utilization for the implementation of basic image processing building blocks and the non-linear diffusion scheme.

4.3.1 eXtreme Processing Platform

The eXtreme Processing Platform (XPP) has been presented in [Baumgarte et al., 2001] as a hardware prototoype and was acknowledged as the fastest 32 bit integer processor at this time. Meanwhile a slightly different internal structure is preferred, but the key features remained the same. We sketch the layout of the device, its programming model and tools. The presentation of the architecture in this section is based on the documentation of XPP provided by PACT. For further details we refer to the company's web site [PACT].

4.3.1.1 Technology

Figure 4.29 on the next page shows a diagram of the XPP architecture. It is a regular array made up of IO elements and a Configuration Manager (CM) and PAEs, i.e. Arithmetic and Logic Units (ALUs) and RAM elements. Between rows of PAEs there are horizontal data and event lines which can be configured to conect to the PAEs in the row below and above. Vertical connections are peformed by FREGs and BREGs. Each PAE consists namely of three parts: a FREG, a BREG, and a main part which is an Arithmetic and Logic Unit (ALU) or a Random Access Memory (RAM) block (Figure 4.30 on the facing page). Data lines have a typical word width of several bytes, while event lines carry just a few bits.

So the internal structure is similar to a FPGA (cf. Section 3.2.4 on page 93), but the reconfigurability resides on a coarser level. It has the advantage that the hardwired ALUs operate on data words more efficiently, that configurations are much smaller allowing fast *dynamic reconfigurability*, and that the overall design and debugging are more transparent and less complicated. On the other hand the coarse grain functionality loses the low level flexibility of the available resources: the data width is fixed wasting resources for smaller number formats, *pipelining* (cf. Section 3.2.1 on page 84) must be performed on coarser elements, computational resources cannot serve as memory, applications specific commands are not available and we have the same design clock for the whole chip. But the great efficiency of massive *parallelism in depth* and *in breadth* (Section 3.2.2 on page 87) and the flexibility to exchange resources between these different types of parallelism remains.

In the simulator one can change the arrangement of the array and the different parameters like number and functionality of ALUs, RAMs, IOs or number and width of data and event lines.

Figure 4.29 Diagram of the XPP architecture, courtesy of PACT [PACT]. We see one possible arrangement with two types of Processing Array Elements (PAEs): Arithmetic and Logic Units (ALUs) and RAM blocks.

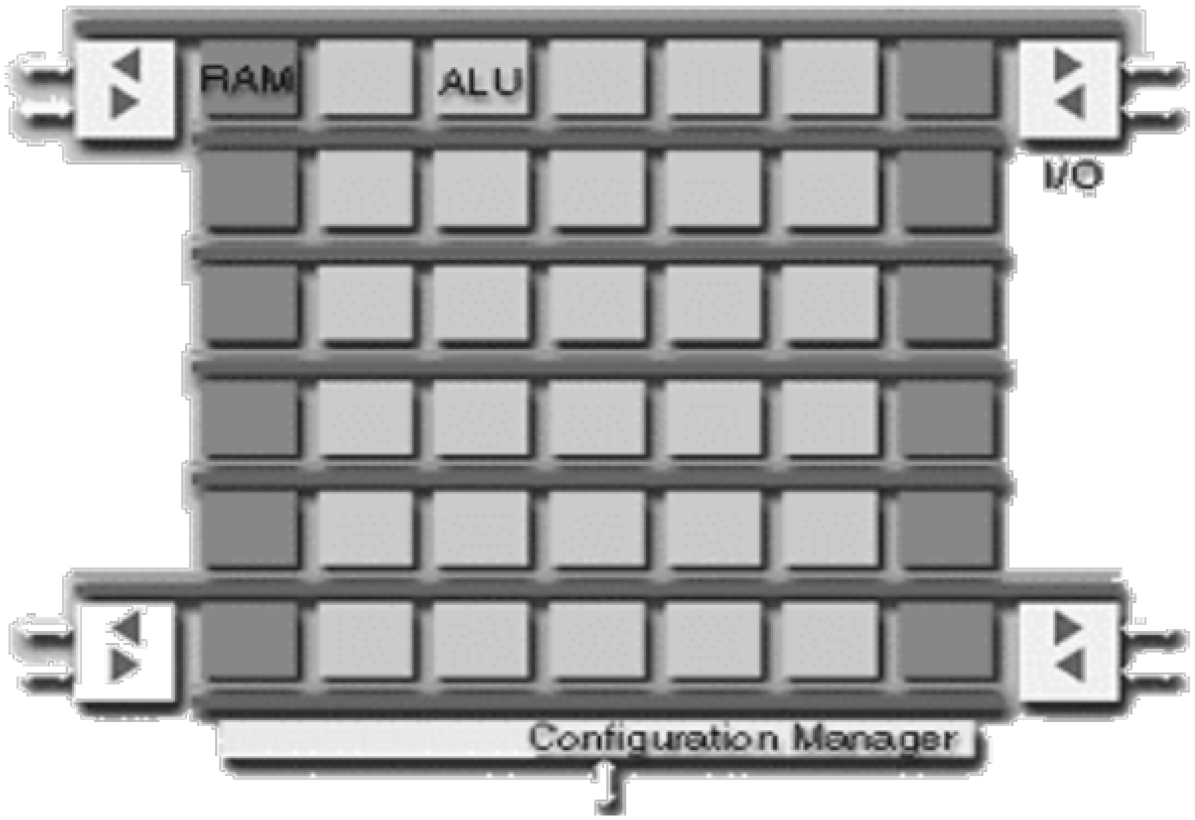
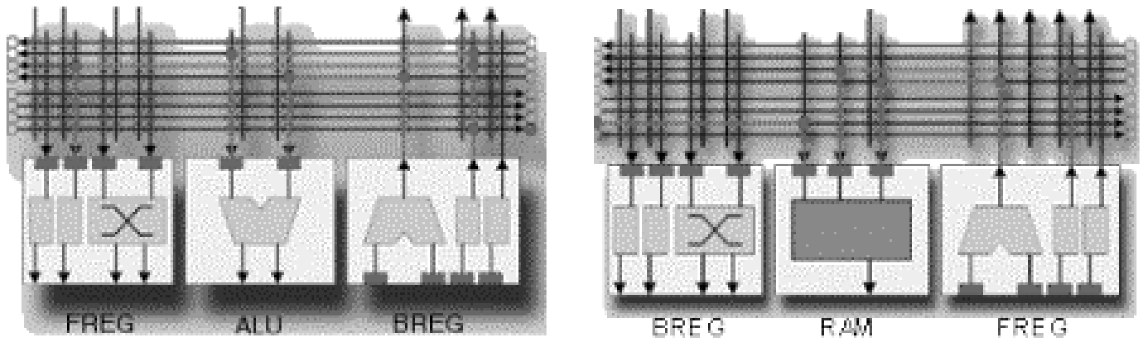


Figure 4.30 Processing Array Elements of the XPP, courtesy of PACT [PACT]. Both types have a Forward Register (FREG) and a Backward Register (BREG) which are used for vertical routing and minor computations. The central parts offer either arithmetic and boolean functions (ALU) or a block of memory (RAM).



In a fabricated chip this is not possible, so we worked with a fixed layout in the simulator, which has later been made available in a chip XPP64-A1 [PACT]. The layout consists of an 8x8 array of ALU PAEs with 8 RAM PAEs and two IO elements on the left and right. The data width is 24 bit, event lines carry 1 bit. The routing capacities are set to 8 horizontal data and 6 event lines in each direction and 4 vertical data and event register routings in each FREG and BREG.

The size of the RAM blocks is 2^9 words. The RAM and ALU PAEs occupy the same area, so the relatively small memory size examplifies how cheap computational resources are in comparison to internal memory. The design of stream processors (Section 3.2.4 on page 93) also relies on this fact [Khailany et al., 2003]. We obtain an architecture with a lot of computational power, but because of its multiple IO elements, the XPP can also deal well with memory intensive problems of smaller *computational intensity*.

Now, we look at the individual components in more detail.

- Processing Array Element (PAE) (Figure 4.30 on the page before)
  - ALU
    The ALU can perform arithmetic and boolean operations like addition, multiplication, shifts, boolean AND/OR, comparisons, max/min. The operations come in at least two versions, assuming either signed or unsigned integers as operands. This means that there is no explicit number format, one rather decides for each configuration how to interpret the data lines.
  - RAM
    The memory can be read and written to by specifying an address and the input word. Additionaly a First In First Out (FIFO) mode is available, which is particularly useful for caching and delaying of a data stream.
  - FREG
    Besides vertical routing capabilities, FREGs can also execute conditional counters, assignments or discards triggered by event signals. This is very important for generating, merging and eliminating of data elements in streams.
  - BREG
    BREGs provide vertical routing channels and can also be used for data addition, shifts and a lookup table for events.

  Each of the above operations performs in one clock cycle, only the BREGs can operate and route data through without a logic delay. It should be noted that the functionality of an ALU PAE is very similar to the functionality of the PEs in the Fragment Processor (FP) of DX9 GPUs (Section 4.1.2.3 on page 124). Here, we only miss the special units which perform the reciprocal and trigonometric computations which make more sense for the floating point format in GPUs. With 128 floating point operations per clock cycle in the FP in the newest GPUs(DX9+), they have also reached a similar level of parallelism. The maximum number of operations in a 8x8 ALU arrangement of the

XPP is namely 64 multiplications (ALUs) and 64 additions (BREGs). The XPP has the advantage of programmble routing between the PE, but this in turn complicates the programming model.

- Input/Output (IO)
  Similar to the internal RAM, the IO elements, which communicate with the external memory, can be either used in a random access mode by providing a memory addresses or a streaming mode with continuous in- or out-flow of data. From our detailed discussion of memory access (Section 3.1 on page 71) and the many delays associated with it, it is clear that these operation do not have a *latency* of one clock cycle. But in the streaming mode we can obtain a very high *sustained bandwidth* (Section 3.1.1.2 on page 76), ideally producing only one initial latency. For the random access mode this is more difficult, but the DSB computing paradigm allows for more flexibility in hiding the incurred latencies. Although the memory will be accessed with random addresses, they usually can be generated well in advance of the actual processing of the data, so that the latency is hidden by the ongoing computation.

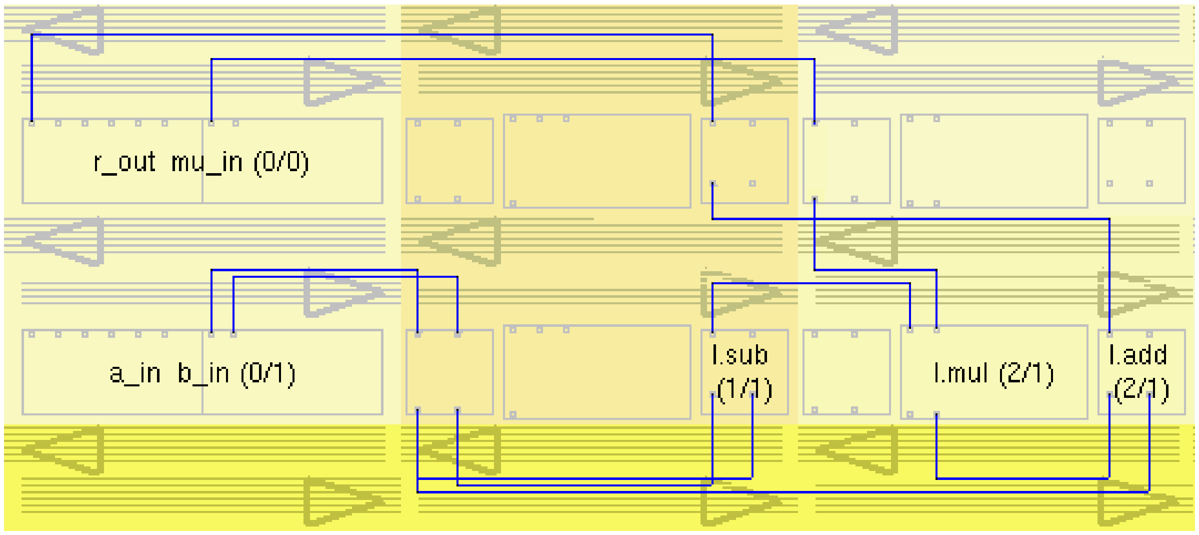
- Configuration Manager (CM)
  The CM can be triggered by event lines to select and load a new configuration for the array, i.e. the array is capable of self-reconfiguration, but the configurations themselves are static and must have been specified in advance. One of the great advantages of the XPP architecture is its *dynamic reconfigurability*, i.e. the ability to use each PAE immediately after its configuration. This means that during the configuration of a problem solver, the PAEs already work on the solution, although the configuration is not finished yet. In particular when changing configurations this feature hides most of the latencies incurred by the configuration process. The XPP supports even ***differential reconfigurability***, i.e. to achieve a faster reconfiguration only the difference to the current state must be specified, e.g. the constant coefficient of a filter.

The reason why the dynamic reconfigurability works without losing any data, e.g. when the data stream reaches a PAE which has not been configured yet, is the automatic packet handling with self-synchronizing PAEs. Data words and event signals are sent as one packet per clock cycle and internal protocols ensure that no packets are lost. A self-synchronizing PAE executes its configured operation only if all input data and event packets required for that operation are available, otherwise it waits for the necessary packets to arrive. This makes programming and debugging of configurations much easier since data is never lost unintentionally and the morphware never executes on undefined data. Overall, we identify the following main virtues of the XPP architecture:

- Massive parallelism.
- Dynamic self-reconfigurability.
- Self-synchronization of data packets

We use only one array for our implementations. But the architecture also scales well and designs with several processing arrays controlled by a hierarchy of CMs are possible. In fact, the first presented prototype [Baumgarte et al., 2001] was an arrangement of two 8x8 arrays.

Figure 4.31 Visual representation of the XPP configuration from Listing 4.3 on the facing page by a screenshot of the `xmap` tool. The left column contains the IO elements, the midlle the RAM PAE and the right the ALU PAE. For this example a tiny XPP array has been defined, usually 6 rows lie between the upper left and the lower left IO element.
The configuration implements a linear interpolation $a + \mu(b - a)$. For this purpose the input slots of the IO elements (`a_in, b_in, mu_in`) are appropriately connected to the slots of the subtracter (`l.sub`), multiplier (`l.mul`) and adder (`l.add`). Finally, the output slot of the adder is attached to the output slot of an IO element (`r_out`).



4.3.1.2 Programming

The NML, a type of structural Hardware Description Language (HDL), is used to write configurations for the XPP. It specifies the *configware* (Table 3.2 on page 85) and may also specify the *flowware* for the problem, depending on the mode in which the IO elements are used. In the stream mode the data stream assembly and storing must be performed by a *data sequencer* outside of the chip. In the random access mode, one has to configure several PAEs as address generators within the array to perform the task of the *data sequencer*.

A configuration of a PAE in NML is given by the operations to be executed and the connections of the inputs and outputs to constants, other PAEs or the IO elements (see Figure 4.31 and Listing 4.3 on the next page). Because of the lossless packet handling, a set of correctly connected and configured PAEs will deliver the correct result for the overall computation irrelevant of the actual timings of the operations. So a formula needs only to be split into operations which can be executed on the PAEs, e.g. $a + \mu(b - a)$ needs two BREGs (SUB, ADD) and one ALU (MUL) connected as ADD(a, MUL(μ, SUB(b,a))). The first operand a arrives very quickly at the input of the ADD BREG, but the self-synchronization gurantees that it does not execute the addition before the result from the multiplication arrives. In a unsynchronized circuit the addition would execute in each clock cycle and thus combine a with

Listing 4.3 Example configuration of the XPP in the Native Mapping Language (NML). The configuration implements a linear interpolation $a + \mu(b - a)$, see Figure 4.31 on the preceding page for a visualization of this configuration. NML keywords are printed in bold, italic emphasizes the predefined names of the input and output slots, e.g. line 11 connects the first input slot of the multiplier (*A*) to the first result slot of the subtracter (*X*).
Lines 1-19 only define the LERP module, the actual object instances and their placement (@$x,$y) is given in the module main. Lines 23-28 set the names of input and output lines to the IO elements, lines 30-34 connect these to the slots of an instance of the LERP module. Because only object functionality, connections and placement are specified, this is not a temporal but a spatial programming model, where the order of the statements bears no meaning.

```
 1 MODULE LERP(DIN a, b, mu, DOUT X)     // linear interpolation
 2 {
 3   X= add.X                            // module output
 4
 5   OBJ sub: BREG_SUB @ 0,0 {           // subtracter b-a
 6     A= b
 7     B= a
 8   }
 9
10   OBJ mul: MUL @ 1,0 {                // multiplier mu(b-a)
11     A= sub.X
12     B= mu
13   }
14
15   OBJ add: BREG_ADD @ 1,0 {           // adder a+mu(b-a)
16     A= mul.Y
17     B= a
18   }
19 }
20
21 MODULE main
22 {
23   OBJ a_in: DIN0 @ $0,$1              // a stream in
24   OBJ b_in: DIN1 @ $0,$1              // b stream in
25   OBJ mu_in: DIN0 @ $0,$0             // mu stream in
26   OBJ r_out: DOUT1 @ $0,$0  {         // r stream out
27     IN= l.X
28   }
29
30   OBJ l: LERP @ $1,$1 {               // module instance
31     a= a_in.OUT
32     b= b_in.OUT
33     mu= mu_in.OUT
34   }
35 }
```

some undefined value and later the multiplication result with some other undefined value. In un-synchronized circuits the timings of the operations are therefore crucial to its functionality and only a careful placement of delays delivers the correct result (cf. Figure 3.4 on page 86). For the XPP this is not necessary, but an automatic or hand-made data path balancing with delays should also be applied in a second step to maximize the *throughput* of the design.

If different configurations have been specified, we need one additional source file which tells the CM which of them should be configured at start up, which will follow or may be selected by event signals from within the array.

Although far less details have to be specified in NML than in a typical HDL for FPGAs, it is still a very low level language. PACT envisions a development environment in which a standard HLL like C is splitted automatically into sequences of well vectorizable code which are translated into XPP configurations and inherently sequential code. In a CSoC both the processing array and a ISB processor reside. The processor executes the sequential code and selects the configurations for the array, which perfroms the computationally demanding parts of the code. First steps in this directions have been undertaken [Becker et al., 2003; Cardoso and Weinhardt, 2002]. There is a growing demand for such flexibe CSoCs [Becker, 2002], but it is known from similar approaches in Reconfigurable Logic (RL) that an automatic extraction and parallization of code from a sequential language is very difficult and a naive approach often results in bad resource utilization. Our implementations were coded directly in NML to exploit the full processing power of the relatively few PAEs.

The NML programmer is supported by the XPP development suite (XDS), which consists of three programs: `xmap, xsim, xvis`. The first maps NML onto the array, i.e. places the PAE configurations and routes the inputs and outputs. The second simulates the entire execution in a clock accurate manner. The third visualizes the computation process. Unlike a FPGA, where finding an error by observing the computation on thousands of lookup tables (LUTs) is quite impossible, here the visual presentation allows to trace individual data packets through the array and thus hopefully identify the error. Also the development cycle which runs through these three steps is much shorter than for FPGAs, where the place&route step easily consumes hours after each modification. The only problem when working with XDS, which also applies to other hardware simulation tools, is that the simulator requires a lot of time for the clock cycle exact modifications of all states in the XPP. So tests could only be run on very small data sets and not entire images. The simulation also does not cover all aspects of an implementation, since the processing with the XPP usually depends on external memory access. But the now available PCI board including the XPP64-A1, SRAM memory and a micro-controller offers a fully realistic development environment.

4.3.2 Computations

We discuss the data-flow for image processing and some computational aspects of the architecture. The coarse grain architecture offers many advantages but also some restriction in comparison to fine grain FPGAs (Section 4.2.2 on page 159).

4.3.2.1 Data-Flow

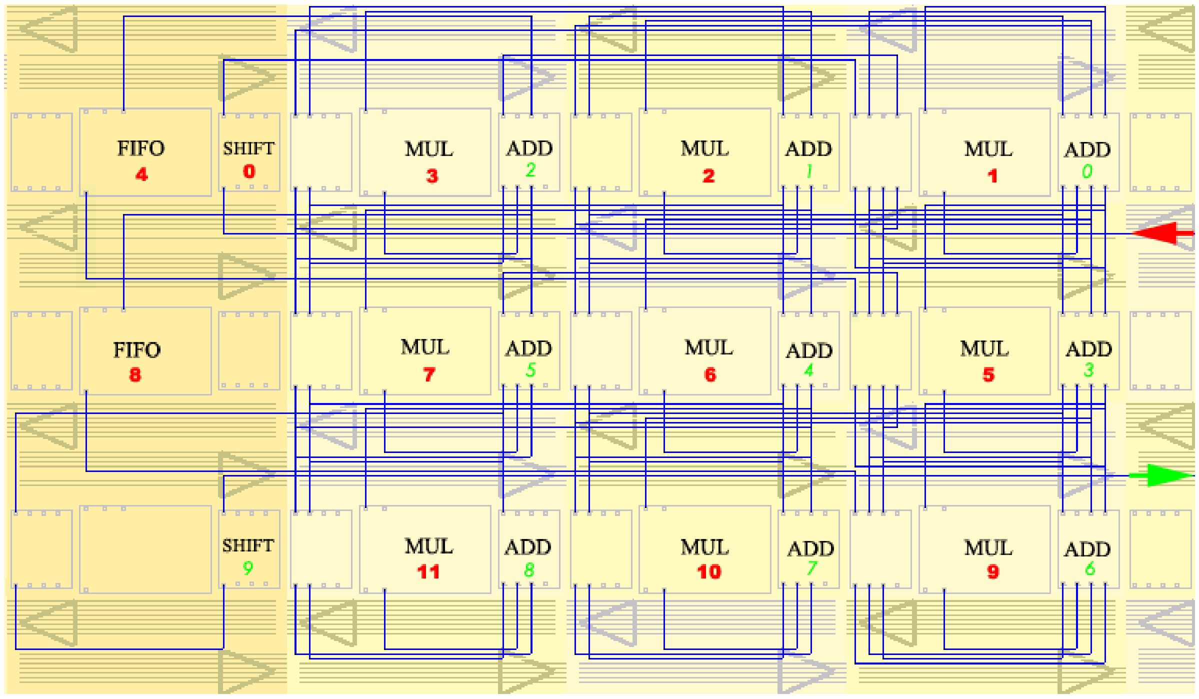
In general the data-flow is very similar to the FPGA setting in Section 4.2.2.1 on page 159. But unlike the FPGA case where we can also trade computational resources for local memory, here we must deal with a predefined ratio of ALUs to RAMs. The guiding idea is again to efficiently utilize the available memory bandwidth and all computing resources of the device, since bandwidth shortage is often the bottleneck (cf. Section 3.1 on page 71) and *spatial computing* translates for reconfigurable architectures directly into performance gains (Section 3.3.1.4 on page 100).

Because of the two dimensional arrangement of the PAEs with the RAMs in the side columns we can have a quadratic stencil of PAEs operate on a corrsponding quad in the image. Figure 4.32 on the next page shows a 3x3 filter arranged in this way. The simple example gives a very good idea of how the local data-flow in image processing is usually set up. The spatial arrangement of the ALUs corresponds to a local neighborhood in the image and the RAM elements serve as line caches for maximal data reuse. More complex schemes operate similarly adding the simultaneous proceessing of several streams and their interference.

These local data-flow structures are simpler to achieve on the XPP as on FPGAs, because the configuration of the RAM block requires only a few lines to use it as a FIFO with a specified delay. Also, there is a clear correspondence between the arithemtic operations and their functional units on the chip. Moreover, the configuration is more flexible, because with *differential reconfigurability* one can, for example, quickly change the constant weights for the multipliers. For a fully adaptive filter the weights have to change constantly, which can be achieved by defining the coefficients of the multipliers with previous computation results (see Section 4.3.3.1 on page 176).

The global data-flow is governed by the data sequencer which assembles and writes the data streams from and to external memory. From the description of the local data-flow in gather operations above, it is clear that the larger the image and the stencil on which we operate, the more local memory is needed for the caching. If the local memory suffices then we may stream the entire image at once and no explicit address generation is required. But if the local memory is too small we must split the image into smaller tiles and process these tiles individually. Unless a host processor organizes the data stream in a tile order, we need to invest some of the PAEs for the generation of the appropriate addresses in the array. These address generators are mainly counters which have to resolve different cases concerning the interior borders between tiles and the exterior border of the entire image. Because boundary conditions for the image domain can be realized by duplicating border pixels, it is often more convenient to generate these ghost cells with the address generator than to disturb the regular structure of the parallel computation with conditional statements. However, internal address generators should only be applied if memory bandwidth is not the limiting factor, because the IO elements have a doubled bandwidth in the stream mode in comparison to the random access mode.

Figure 4.32 Implementation of a general 3x3 filter in the XPP. A screenshot of the `xmap` tool after routing and placement with added information is shown. The filter is applied by processing an image as a stream of pixels in row order. The pixel stream enters with the upper line from right. The **bold numbers** give the order in which the stream traverses the functional units. The multipliers MUL which apply the weights to the pixels are arranged in the same way as the 3x3 quad in the image they operate on. The *light italic* numbers give the order by which the result stream as the sum of the multiplications is generated. It is ouput by the lower line to the right. Because of the fully pipelined arrangement with local FIFOs caches, in each clock cycle one input pixel enters and one result pixel leaves this configuration.
The pixel values are shifted in the begining (SHIFT **0**), such that the integer multiplications correspond to the desired fixed point number format. Subsequently the pixels traverse the multipliers of the first row (MUL **1-3**) and then are cached in the FIFO (**4**). The delay in the cache is such that, when the pixel leaves the cache and is processed by MUL (**5**), the result of the multiplication is added by ADD (*3*) to the partial sum (coming from ADD *2*) of the corresponding weighted pixels from the image line above. After passing the second row of multipliers (MUL **5-7**) the pixel stream is delayed again in the same manner. So for any multiplier and adjecent adder the pixel and result streams are synchronized, so that the weighted sum of the 3x3 neighborhood is accumulated one by one. Before being output, the final result is shifted back to the original format (SHIFT *9*).
The extension to larger filters operating in the same fashion is obvious. Here in each clock cycle 9 multiplications and additions are performed, but only one pixel is read, minimizing the required memory bandwidth. The numbers apply to a general filter, a seperable filter performs the horizontal and vertical calculation separately, requiring only 6 multiplications and 4 additions.



4.3.2.2 Operations

The available operations in the individual components of the XPP have already been discussed in Section 4.3.1.1 on page 166. All operations compute on intergers of a fixed width. The ALU PAE has two inputs and outputs. The two ouputs allow to deliver the exact result of a multiplication, i.e. a high and a low word, and together with inclusion of carry over functionality in the addition it could also be used to support wider data formats.

To obtain a *fixed point number* format from the integers, we have to shift the numbers appropriately. For linear computations this is ideally done at the entrance of the data stream only once for each element (cf. Figure 4.32 on the facing page). Whether the operands in operations are treated as signed or unsigned integers is determined by the choice of the operation at the ALU. So with the 24 bit integer format we have a signed fixed point format of the same size as the mantissa plus sign of the s23e8 floating point format used in DX9 /+ graphics hardware. In general the missing *exponent* is not crucial becomes we have designed the numerical schemes to operate on similar number scales.

But without *floating point numbers* ALUs non-linear functions cannot be represented accurately on the XPP. The RAM blocks can be used as very quick LUTs, but they allow only 9 address bits. A masking mechanism can be used to catenate several RAM blocks to a comon LUTs, but even all of them would result in just 13 bit precision. GPUs do not offer 24 bit wide LUTs either, but their predefined non-linear functions like reciprocal, power and trigonometric functions can represent the results accurately in the floating point format. The 8 bit precise addressing of a LUT in the XPP is not that bad for one dimensional non-linear functions, since one can improve the accuracy by linear interpolation, but for two and more dimensional functions, e.g. for vector normalization (cf. Eq. 4.8 on page 135), one would have to use a LUT in external memory. After all GPUs do exactly this in dependent texture accesses. In a FPGA one could encode a LUT of exactly the required minimal precision, but for two and more dimensions this would still consume too many resources and again external memory would have to be used.

So if floating point operations are important for the envisioned application area, it makes sense it include corresponding hardwired ALUs into the chip. In case of the XPP one could simply replace the fixed point PAEs by floating point PAEs. But such a decision would have to be made before chip fabrication.

4.3.3 Non-Linear Diffusion

We configured the XPP with a semi-implicit solver for the *non-linear diffusion* model for image denoising. The model is the same as in Section 4.1.4 on page 132, where a graphics hardware implementation has been presented, but here we restrict ourselves to the isotropic non-linear diffusion case.

The company site [PACT] presents some examples of basic signal and image filters, but because the architecture is fairly new and has been made available in hardware only recently implemenations for it are still sparse. An overview of other Reconfigurable Computing (RC) devices used for Digital Signal Processing (DSP) and in particular image processing can be found in [Sueyoshi and Iida, 2002].

4.3.3.1 Implementation

We use the semi-implicit scheme

$$\begin{aligned} A'^{+}(\bar{U}^n_\sigma) \cdot \bar{U}^{n+1} &= \bar{U}^n \\ A'^{+}(\bar{U}^n_\sigma) &:= \mathbb{1} + \frac{\tau^n}{h^2} L'[\bar{U}^n_\sigma]\,, \end{aligned} \tag{4.12}$$

and apply a quantized version of the Jacobi solver (Eq. 2.39 on page 45) to solve the linear equation system

$$\begin{aligned} F(\bar{X}^l) &= D'^{-1}(\bar{R} - (A'^{+} - D')\bar{X}^l), & D' &:= \mathrm{diag}(A'^{+}) \\ &= \left(\mathbb{1} + \tfrac{\tau^n}{h^2} L'_D\right)^{-1} \left(\bar{R} - \tfrac{\tau^n}{h^2}(L' - L'_D)\bar{X}^l\right), & L'_D &:= \mathrm{diag}(L')\,. \end{aligned} \tag{4.13}$$

The prime indicates that we deal with the isotropic *non-linear diffusion* in contrast to the *anisotropic diffusion* model (see Eq. 2.1 on page 16). The matrix assembly is simpler in this case. The components of the stiffness matrix $L'[\bar{U}^n_\sigma]_{\alpha\beta}$ are given as a weighted sum of integrations over the elements adjecent to the current node (cf. Figure 2.1 on page 27):

$$\begin{aligned} L'[\bar{U}^n_\sigma]_{\alpha\beta} &= \sum_{E \in E(\alpha)} G'^n_E S'^{\alpha\beta}_E \\ G'^n_E &:= \tilde{g}(\nabla U^n_\sigma(m_E)) \\ S'^{\alpha\beta}_E &:= (\nabla\Phi_\alpha, \nabla\Phi_\beta)|_E\,. \end{aligned} \tag{4.14}$$

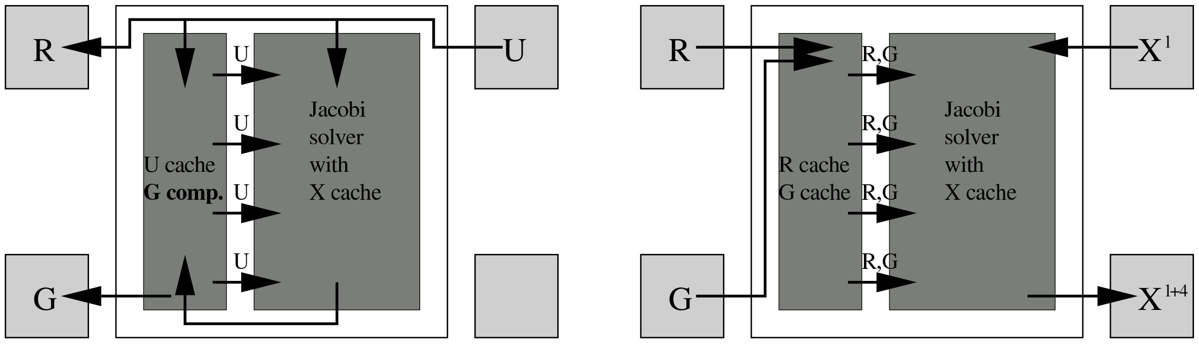
where G'^n_E are the element weights and $S'^{\alpha\beta}_E$ pre-integrated constants depending on the Finite Element (FE) basis functions, they evaluate to

$$S'^{\alpha\beta}_E = \begin{cases} +\frac{2}{3} & \text{if } \alpha = \beta \\ -\frac{1}{3} & \text{if } |\alpha - \beta| = (1,1) \\ -\frac{1}{6} & \text{if } |\alpha - \beta| = (1,0) \\ -\frac{1}{6} & \text{if } |\alpha - \beta| = (0,1) \\ 0 & \text{else} \end{cases} .$$

Obviously, this gives a simpler matrix assembly than the general explicit formula in the anisotropic case (Eq. 2.45 on page 46).

The XPP makes it feasible to run an implicit solver, which avoids the time-step restrictions of the explicit schemes, because we have a higher internal precision (24 bit) and *dynamic reconfigurability*. Fast reconfigurability is important, because the implicit scheme requires

Figure 4.33 Configurations for the non-linear diffusion solver in the XPP architecture. On the left we see the pre-calculation of weights, on the right the linear equation solver for the implicit scheme. The second configuration can execute several times by exchanging the role of input and output elements on the right side of the XPP. Partial reconfigurability allows both configurations to share the large Jacobi solver arrangement and thus reduce the configuration overhead.



two different configurations (see Figure 2.3 on page 64 and Algorithm 4.3 on page 137). The first configuration performs the mollification of the gradient ∇U_σ^n with a *linear diffusion* and then computes the weights $G_E'^n$ for each element. The second configuration executes several iterations of the Jacobi solver (Eq. 4.13 on the preceding page). To minimize the switching between the configuration the Jacobi solver block can be used for both the main linear equation solver and the mollification. Only the connection to the appropriate weights must be changed (Figure 4.33).

The first configuration takes $\bar{U}$ as input and computes $G_E'^n := \tilde{g}(\nabla U_\sigma^n(m_E))$ as output. The mollification uses the Jacobi solver with equal weights for each element. The non-linear function $\tilde{g}$ is evaluated with a lookup table implemented in a RAM PAE. The RAM elements can hold 2^9 different values, so that the evaluation of $\tilde{g}$ cannot take place with full precision. To reduce the resource requirements in the second configuration, we also precompute the inverse factors $\left(\mathbb{1} + \frac{\tau^n}{h^2} L_D'\right)_\alpha^{-1}$ with a lookup table and store them together with the weights in one word, i.e. with a resolution of 12 bit each. The values have to share one word, because there are not enough RAM elements for the necessary caching in the following configuration.

In the second step, the configuration of the Jacobi solver remains the same, only the weights are not constant any more, but based on the results from the previous configuration. In the implementation we connect the coefficients of the general 3 by 3 filter from Figure 4.32 on page 174 to the matrix components resulting from Eq. 4.14 on the facing page. The available computing resources are exploited by a *parallelism in depth* approach, i.e. each configurations performs 4 Jacobi steps in a pipelined fashion by utilizing the RAM PAEs for the caching of the resulting intermediate image lines. As the diffusion process on linear FEs involves three lines for the local computation (see Figure 2.1 on page 27) two lines must be cached in each iteration step. The 8 RAM PAEs on the right side of the array serve this purpose (cf. Figure 4.29 on page 167). Besides the caching of the intermediate iteration vectors $\bar{X}^l$ we must also cache $\bar{U}$, which serves as the right hand side in the equation, and the weigth factors. This is done by the RAM elements on the left side of the array. Because of the restricted number of these elements, for a high number of pipelined iterations, we must pack the weight and inverse factors into one word.

The second configuration is run several times until the residual error becomes sufficiently small. In contrast to graphics hardware which mainly applies hardwired *parallelism in breadth* on separate pipelines and thus cannot easily incorparate a global associative operation on all values of an image into the hardware, a single PAE suffices here to compute the 2-norm of the residual. A similar PAE could also be used to compute the mass defect after the iterations and dither this defect uniformly back into the solution (Eq. 2.52 on page 49) to preserve the overall mass exactly. This would require a long precomputed random list of positions for the image and some more ALU resources for the application of the dither. From a practical point of view this is not really necessary since the higher internal precision (24 bit) keeps the mass defect low.

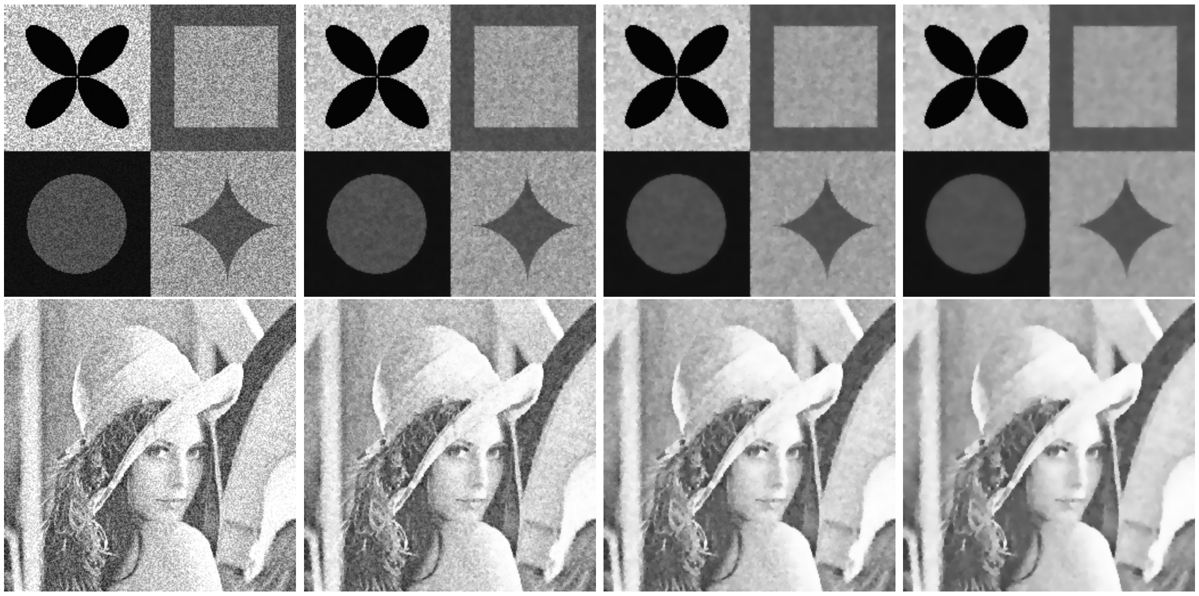
The implicit scheme now allows to specify larger time-steps, but this choice also effects the accuracy of the computation. If we increase $\frac{\tau^n}{h^2}$ in Eq. 4.13 on page 176 significantly, then our fixed-point number format must have many positions before the comma, and we will lose many significant digits in the representation of $\bar{R}$. This is not a problem of the fixed-point format, but a general problem of fixed precision number formats. A floating point format could retain the precision in the representation of both addends, but during the addition the same digits would be lost. So a larger time-step width simply requires a larger number format if we do not want to lose significant information in the process. In practice, this means that the time-step width in implicit schemes becomes restricted by the size of the used number format. For low precison number formats the restrictions for the implicit schemes are almost as severe as for the explicit ones and thus the implict formulation becomes less useful. Therefore, we have favored the explicit schemes for the 8 bit formats in graphics hardware, but here the use of the implicit schemes makes sense.

4.3.3.2 Results

Since the clock cycle exact hardware simulation consumes too much time we could not generate results on entire images with the XPP development suite, but only check the functionality of individual compentents with small data sets. Instead, we present results computed with a self-made C-simulation, which performs the individual steps of the algorithm with exactly the same precision as in the XPP. Figure 4.34 on the next page presents some results.

The advantage of the clock cycle exact simulation is a reliable estimate on the *throughput* of the whole design. The four iterations of the Jacobi solver operate in a fully pipelined fashion, so that after an initial latency we obtain one result pixel in each clock cycle. Assuming a clock frequency of 40MHz (the frequency of the now available hardware development board from PACT) this amounts to more than 2400 Jacobi iterations per second on a 256^2 image or equivalently one iteration takes approx. 0.41ms.

Figure 4.34 Non-linear diffusion as configured on the XPP architecture. The actual computation was performed by a simulator. The result of the second, fourth and fifth time-step is shown. The mollification performed 4 and the implicit solver 8 iterations for each time-step.



It is worth to note that these values are very similar to the performance of the level-set solver on the FPGA (Section 4.2.3.2 on page 163). The difference stems only from the slightly different clock frequency, since both designs apply a similarly pipelined arrangement with a four-fold parallelization of the computation. Although the XPP has approx. 3 times more transistors, the coarse grain appoach is more efficient, since the Jacobi solver requires more operations in each step than the level-set scheme and the XPP computes everything in a twice as wide number format. In comparison to the GeForce3 GPU (8ms for an explicit anisotropic time-step see Section 4.1.4.2 on page 136), which has approximately the same number of transistors but a much higher clock frequency, the XPP does very well, because in each step it can devote almost all of its computing resource to the current task in parallel, while only some of the hardwired PEs could be utilized on the GPU in each intermediate step. This handicap has been greatly reduced in the current DX9 graphics hardware.

A configuration of the XPP with 4 steps for the mollification and 8 steps for the weighted solver could run 5 time-steps (as in Figure 4.34) in real-time (40Hz) on 256^2 images. Certainly, we need to substract the times for the changing of the configurations and the intial latencies in the begining of each iteration, but these times range in the order of $10^3 - 10^4$ cycles and thus would account for at most 10% slowdown. Moreover, the frequency of 40MHz on the hardware development board is fairly low and dedicated image processing solutions based on the XPP could be clocked much higher.

4.4 Comparison of Architectures

There are many competing aspects associated with a choice of hardware architecture, e.g. performance, flexibility, design effort, power consumption, costs. For each of theses aspects there are ideal architectures and the optimization between any two of them can still be handled to a certain extent, but in practice we always want more of them at once. In Section 3.3 on page 97 we have discussed several architectures on a high level with a focus on DSP and in particular image processing. We now continue the discussion refering to the experience with the different implementations.

4.4.1 Instruction- and Data-Stream-Based Architectures

For the implementations we have used data-stream-based (DSB) architectures exclusively. Chapter 3 motivates this choice in detail. It elaborates on the *memory gap* problem, which hits instruction-stream-based (ISB) architectures stronger (Section 3.2.3 on page 92), and discusses the advantages of *spatial computing* over *temporal computing* (Section 3.3.1.4 on page 100). It is legitimate to ask what is the price for the increased performance.

One of key points of DSB architectures is that they require us to disclose more information about the internal structure of the problem. A software program for a ISB processor contains actually very little information about the nature of the problem. From the viewpoint of the processor it is a long inflowing stream of instructions of which it sees only very little ahead. Jump commands can make the stream change rapidly, but despite the short visibility of these events we expect the processor to keep up the pace. The only chance to do this is to speculate on the jump destination and continue the computation in this direction. It is a bit like driving on a densly befogged road and trying to guess which bend comes next. If the guess is wrong one has to brake harshly and re-accelerate again. Naturally, one could go much faster if one saw more of the road ahead. The very fact that the processor has to make guesses about our code reveals how little information we have provided to support a fast execution.

DSB architectures already require us to divulge separate information about the assembly of data streams in form of *flowware* (the road ahead) and the configuration of the processing elements (PEs) in form of *configware* (the engine setup). This distinction is inherent to DSB processing even if the different sources execute on the same device (Sections 4.2.1.2 on page 158 and 4.3.1.2 on page 170), because the inability to provide timely insight into the assembly of data streams (bad road visability) and insufficient computing power of the PEs (weak engine) are independent limiters of performance. Being forced into a dual programming model, one is constantly aware of the importance of balancing the different demands. This further structuring of the problem solver requires an additional design effort by the programmer. But it is reasonable to expect that the programmer has a better overview of the program and the required structuring than the PEs which are concerned with the low-level execution.

There is a continuous trade-off between the grade of problem structuring and obtained performance. The concept of *spatial computing* marks another major step on this scale. The GPU which still applies *temporal computing* differs from the FPGA and XPP in this respect (see Section 4.4.2). By defining the execution in the spatial domain one reveals much more about the number of required resources, their communication and the frequency of configuration changes. The reward is a highly flexible parallelism and a higher resource utilization, but the design effort also increments.

Going further down into smaller PE structures and the explicit synchronization and pipelining of elements, one can further increase performance. But the general concepts of efficient data stream generation and spatial computing already gain the principle advantages. In terms of code reuse and flexiblity it is also not advisable to resolve each problem into low-level structures explicitely. This can be done to some extent by compilers. In fact, the lack of tools which could translate code from a high abstraction level, adhering only to the main DSB concepts of efficient configurations, hinders the faster spreading of DSB architectures. The abstraction entry point seems still to low for many developers. On the other hand, the entry point of standard software High Level Languages (HLLs) is so high and conveys so little information about the problem structure in explicit terms, that attempts to use it as a source for efficient configuration of DSB architectures depend on auxiliary directives in the HLLs. New, DSB oriented HLLs will hopefully fill in this gap in the long run.

4.4.2 Graphics Hardware and Reconfigurable Hardware

All three architectures (Graphics Processor Unit (GPU), Reconfigurable Logic (RL), Reconfigurable Computing (RC)) evolved dramatically in the last 10 years. The RL has a slightly longer success story, but 10 years ago FPGAs were small devices not meant for competing with DSP- or micro-processors. For GPUs the general computing capabilities came up even more surprisingly, and although the functionality demands still stem from the computer game industry, GPU manufacturers now pay attention to the scientific computing possibilities of their products. Finally, RC machines are probably the least present architectures in the market, but a number of ambitious start-ups promotes their advantages and several prototypes and first commercial products are available.

Performance analysis of RL over micro-processors for DSP applications has been thoroughly examined, see for example [Benitez, 2003; Guo et al., 2004; Tessier and Burleson, 2001]. Performance of RC machines on DSP kernels can be found in [Sueyoshi and Iida, 2002; Suh et al., 2003]. For GPUs an overview of image processing implementations is not known to us. The site [GPGPU] contains a thematic collection of abstracts and links to general computing papers on GPUs. In the following we compare the architectures based on the previous sections.

- Graphics Processor Unit (GPU)
  GPUs have improved a lot not only in terms of performance but also in terms of programmability (Section 4.1.1.3). The first implementations (Sections 4.1.3, 4.1.4) had to use a complicated translation from the world of numeric computations to the functionality of the fixed graphics pipeline (Figure 4.1). In particular, the severe restrictions of the number formats required many workarounds (Section 4.1.2.2). First with the pro-

grammability of the Fragment Processor (FP) and the floating point formats in DX9, GPUs gained more attention as solvers of scientific problems. This flexibility allows many more numerical concepts to be implemented (Section 4.1.5 on page 143). The implementation of the registration problem from Section 4.1.5 on page 143 in either RL or RC would presumably deliver higher performance but the design effort would also be much higher.

The flexibility of GPUs grew mainly in the processing elements (PEs), which now offer functions not even supported directly by the ALUs in RC arrays (Section 4.1.2.3). The free programability of the PEs also allows a better resource utilization than the hard-wired, parameter controlled PEs did. In the direct performance comparison of the level-set (Section 4.2.3.1 on page 161) and the non-linear diffusion solver (Section 4.3.3.1 on page 176) the FPGA and XPP win against the older GPUs exactly due to better resource utilization.

GPUs apply a fixed parallelism in breadth given by the number of available pipelines. Parallelism in depth takes place in the different stages of the pipleine and the individual PEs. The pipeline iteself is deep but the PEs have few internal stages. The parallel use of these internal stages is hidden from the programmer and depends on the compiler optimizations in the temporal domain. Because of these restrictions the design of configurations has become fairly easy in comparison to Hardware Description Language (HDL) models for reconfigurable hardware. But the implicit command scheduling and caching of data for reuse requires more control resources not used for the actual computation. Graphics hardware is partly also burdend with compatibility issues which require the realization of the whole graphics pipeline, although the programmable parts could inherit almost all of this functionality. However, in comparison to micro-processors were the L2 cache consume most of the transistors, GPUs devote by far the majority of its resources to computation.

It is a difficult question how the fixed graphics pipeline could be released in favor of more configurable data-flows (Section 4.1.2.1) without sacrificing the simpler programming model. Where RL and RC architectures save on bandwidth requirements by application specific caches and extra long pipelines, GPUs are very bandwidth hungry and access external memory very often. This made them very suitable for problems with low *computational intensity*, but also depent on this expensive fast external memory connection. With DX9+ GPUs the trend of balancing memory bandwidth and PE requirements started to shift towards a higher number of PEs. Figure 4.25 illustrates how in this case parallelism in depth must be used to increase performance. GPUs will probably not adapt to the spatial computing for this, but the hierarchical stream caches of stream processors (see Section 3.2.4 on page 93) could be an option.

Recapitulating, we can say that GPUs are currently the easiest accessible DSB architecture. They lack programmable parallelism in depth and data-flow flexibility but compensate this with fixed parallelism in breadth and wide memory buses at high clock frequencies. The high clock frequencies still cannot compete with the utter parallelism in large RL or RC devices and they render low power applications unattainable. But the mass market implies a fantanstic price-performance ratio and standard APIs ensure compatibility.

- Reconfigurable Logic (RL)

  RL concentrated for a long time on fully homogenous fine grain arrays (Figure 4.22). This approach guarantees utmost flexibility in the design of the circuits. Arbitrary operations on application specific number formats with *variable precision arithmetic* can be realized (Section 4.2.2.2). There is even no clear distinction between PEs and memory since the Configurable Logic Blocks (CLBs) can server as both (Section 4.2.2.1). So all parallelization options can be fully exploited. The performance advantage of the FPGA (Section 4.2.3.2) against the GPU implementation (Section 4.1.3.2) for the level-set equation grew out of the multiple parallelization options.

  The price for the universitality is the overhead associated with the configuration logic. The same design in an ASIC consumes four to six times less transistors, which, beside the costs and performance, is often crucial in terms of power consumption. So RL depend on a efficient use of their resources. The approximations applied in Section 4.2.3.1 had the aim to replace resource hungry operations (Table 4.4) by simpler ones. But our use of CLBs as memory, for example, consumes much more transistors than a dedicated memory block. Because these situations appeared in many implementations, modern FPGAs have many hardwired multipliers of a certain width and dedicated RAM blocks. Some FPGAs embed even entire processor cores. So one sacrifies the full reconfigurability but gains performance and lower power consumption for all applications which would otherwise implement the hardwired functionality in the CLBs. By further mixing reconfigurable logic, DSP-processors and specialized hardware components on a Configurable System-on-a-Chip (CSoC), one obtains a continuous space expanding from fully general to very specific solutions.

  Naturally, this heterogenounes environement makes the design of configuration even more demanding than before (Section 4.2.1.2). But the design tools have matured and synthesis of configurations from dialects of common HLLs now delivers respectable results. Hardwired blocks also reduce the inherent problem with fine grain architectures of large configuration bitstreams. The large bitstreams make it difficult to change the functionality in the FPGA quickly. This is not crucial if the entire application fits into the FPGA as in Section 4.2.3.1, but for large projects divided into several parts, or different applications requiring fast transitions it means a performance penalty. *Partial reconfigurability* helps to hide latencies in these cases and the hardwired parts need far less configuration data.

  FPGAs are still the most flexible DSB architectures. The parts with fixed functionality are usually so general (multipliers, RAM blocks), that most configurations benefit from them. Performance in comparison to other architectures depends strongly on the application, i.e. if its components can be easily implemented by the predefined PEs in GPUs or RC arrays or not. Power consumption per transistor is high, but the high flexibility allows the use of smaller devices and reconfigure them repeatedly as needed. Compatibility among different systems is hardly an issue, but sufficiently abstract HDL code can be synthesized on different devices. Finally, a mass production secures moderate costs of FPGAs and a continuous evolution of design tools.

- Reconfigurable Computing (RC)

  RC architectures give a different answer than the inhomogenous CSoCs based on FPGAs to the overhead associated with bit-level reconfigurability. They homogenously specialize the data paths to fixed widths and lift the operational configurability to the arithmetic level (Section 4.3.1.1). This immensely reduces the configuration stream, which allows *dynamic reconfigurability* and greatly simplifies the configuration design and debugging (Section 4.3.1.2). However, current design tools for RC lag behind the RL development, such that in practice the design effort is not necessarily smaller.

  While RC systems retain the numerous opportunities for parallization, the fixed data path width and operation set miss some of the application specific optimizations (Section 4.3.2.2). It can even render an implementation totally unfeasible if the simulation of a special operation would consume too many resources in the RC chip. In this respect RC systems are similar to the GPUs or DSP-processors which must be defined with a operation set suitable for the envisioned application area before fabrication. But if the set fits the application then very high performance and low power consumption similar to an ASIC can be achieved with RC. The superior performance of the slowly clocked XPP in case of the non-linear diffusion solver (Section 4.3.3.2) against the GPU implementation (Section 4.1.4.2) examplifies the parallelization advantage.

  RC machines offer a very good compromise between high performance, low power consumption and flexibility. If the internal PEs contain the required functionality for a certain application area, then they can provide the best resorce utilization in this area. But because fabrication of commercial products started only few years ago, there are still several drawbacks. The available devices are relatively small and expensive. There are several competing RC systems with different interal structures on the market, so compatibility will not be an issue for a long time either. Also, the development suits are not as comfortable and powerful as for other devices, resulting practically in high design efforts. But these shortcomings seem to be of temporary nature.

4.4.3 Efficiency and Abstraction

The presented hardware architectures are very fast developping technologies. They approach the market of scientific computing from such different fields as computer games, rapid-prototyping, telecommunication. They are drawing nearer to each other by adopting similar concepts. GPUs introduce more configurability into the graphics pipeline, FPGAs partly coarsen their structure similar to RC machines. One even observes a weak convergence of architectures on a larger scale. Micro-processors include more parallel processing elements (PEs) and some DSP processors contain reconfigurable islands. Different hybrids made up of DSP-, micro-processors and reconfigurable hardware evolve as CSoCs.

This convergence will certainly not lead to one ideal architecture. The requirements of applications are too different for this to happen. But market forces will prefer the consolidation to few designs suitable for mass production. Since PEs are always arranged in the spatial domain, an important question is whether High Level Languages (HLLs) can be developped which efficiently exploit this spatial parallelism on the raw PEs. Current Hardware Description Languages (HDLs) are still quite inaccessible to high level programmers. But more abstract approaches require additional control logic around the PEs, which reduces the efficiency.

Our results in the field of image processing suggest that RC arrays offer the most convincing combination of performance and flexibility. But the development tools lag behind the FPGA design suits, therefore FPGAs have already occupied many areas of the DSP market. Because the market is often ready to sacrifice even more hardware performance for ease of programming, graphics hardware with its easy access to the data-stream-based (DSB) processing could finally win the race. It does not have a perfect resource utilization, but many may be willing to pay this price to avoid dealing with an unfamiliar spatial programming model. It is clear that the more we uncover the structure of an algorithm to the hardware, the faster we can run an implementation. Most current software reveals so little about the structure of the implemented algorithms that its performance is very poor. We cannot continue in this way. But the extreme of always operating on the lowest implementational level is not an option either. We are really in need of HLLs which will account for the main hardware characteristics without sacrificing system compatibility and algorithmic abstraction. This ideal balance between hardware efficiency and program abstraction is yet to be found.

Bibliography

J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot. Applications tuning for streaming SIMD extensions. *Intel Technology Journal Q2*, 1999. 98

S. T. Acton. Multigrid anisotropic diffusion. *IEEE Transactions on Image Processing*, 7: 280–291, 1998. 43

J. Alakarhu and J. Niittylahti. DRAM performance as a function of its structure and memory stream locality. *Microprocessors and Microsystems*, 28(2):57–68, Mar 2004. 71

L. Alvarez, F. Guichard, P. L. Lions, and J. M. Morel. Axioms and fundamental equations of image processing. *Arch. Ration. Mech. Anal.*, 123(3):199–257, 1993. 19

L. Alvarez, J. Weickert, and J. Sánchez. Reliable estimation of dense optical flow fields with large displacements. *International Journal of Computer Vision*, 39:41–56, 2000. 24

AMD. AMD Athlon 64 Processor. http://www.amd.com/athlon64/, 2004. 81

P. Atanassov and P. Puschner. Impact of DRAM refresh on the execution time of real-time tasks. In *Proc. IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34, Dec. 2001. 77

M. Baker/Ed. Cluster computing white paper. Technical report, IEEE Computer Society's Task Force on Cluster Computing (TFCC), Dec. 2000. 99

E. Bänsch and K. Mikula. A coarsening finite element strategy in image selective smoothing. *Computing and Visualization in Science*, 1:53–63, 1997. 44

C. W. Barnes, B. N. Tran, and S. H. Leung. On the statistics of fixed-point roundoff error. *IEEE Trans. Acoustics, Speech and Signal Processing*, ASSP-33(3):595, 1985. 35

P. H. Bauer and L.-J. Leclerc. A computer-aided test for the absence of limit cycles in fixed-point digital filters. *IEEE Transactions on Signal Processing*, 39(11), 1991. 36

V. Baumgarte, F. May, A. Nckel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'2001)*, Las Vegas, USA, 2001. 166, 169

J. Becker. Configurable systems-on-chip: Commercial and academic approaches. In *9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002*, Dubrovnik, Croatia, Sep. 2002. 84, 95, 172

J. Becker, A. Thomas, M. Vorbach, and V. Baumgarten. An industrial/academic configurable system-on-chip project (CSoC): Coarse-grain XXP/Leon-based architecture integration. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 11120–11121, 2003. 100, 172

G. Bell. All the chips outside: The architecture challenge. In *International Symposium on Computer Architecture (ISCA) 2000*, 2000. 84

D. Benitez. Performance of reconfigurable architectures for image-processing applications. *Journal of Systems Architecture*, 49(4-6):193–210, 2003. 101, 159, 181

J. Bier. Processors with DSP capabilities: Which is best? In *Proceedings Embedded Systems Conference (ESC) 2002*, 2002. 100

G. Bohlender. Literature on enclosure methods and related topics. Technical report, Universität Karlsruhe, 1996. 30

J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proceedings of SIGGRAPH 2003*, 2003. 133, 143

K. Bondalapati and V. K. Prasanna. Reconfigurable computing systems. In *Proceedings of the IEEE*, July 2002. 93, 100, 102, 156

F. Bornemann, D. Laurie, S. Wagon, and J. Waldvogel. *The SIAM 100-Digit Challenge: A Study in High-Accuracy Numerical Computing*. Society of Industrial Applied Mathematics (SIAM), Philadelphia, 2004. 26

S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, 1996. 93, 101, 156

J. M. P. Cardoso and M. Weinhardt. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. *Lecture Notes in Computer Science*, 2438:864–??, 2002. ISSN 0302-9743. 172

V. Caselles, F. Catté, T. Coll, and F. Dibos. A geometric model for active contours in image processing. *Numer. Math.*, 66, 1993. 21

F. Catté, P.-L. Lions, J.-M. Morel, and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM J. Numer. Anal.*, 29(1):182–193, 1992. 19

E. Cerro-Prada and P. B. James-Roxby. High speed low level image processing on FPGAs using distributed arithmetic. In R. W. Hartenstein and A. Keevallik, editors, *Field-Programmable Logic: From FPGAs to Computing Paradigm, Proceedings FPL 1998*, pages 436–440. Springer-Verlag, Berlin, Aug/Sep 1998. 101

G. E. Christensen, S. C. Joshi, and M. I. Miller. Volumetric transformations of brain anatomy. *IEEE Trans. Medical Imaging*, 16, no. 6:864–877, 1997. 24

U. Clarenz, M. Droske, and M. Rumpf. Towards fast non-rigid registration. In Z. Nashed and O. Scherzer, editors, *Contemporary Mathematics, Special Issue on Inverse Problems and Image Analysis*. AMS, 2002. 23, 24, 150

R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In *Proceedings of the Design Automation and Test in Europe 1999*, pages 271–276, 1999. 36

P. Colantoni, N. Boukala, and J. da Rugna. Fast and accurate color image processing using 3d graphics cards. In *Proceedings Vision, Modeling and Visualization 2003*, 2003. 133

K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2), 2002. 101, 102, 156

L. Corrias, M. Falcone, and R. Natalini. Numerical schemes for conservation laws via Hamilton-Jacobi equations. *Mathematics of Computation*, 64:555–580, 1995. 21

L. D. Coster, M. Engels, R. Lauwereins, and J. Peperstraete. Global approach for compiled bit-true simulation of DSP-applications. In *Proceedings of Euro-Par'96*, volume 2, pages 236–239, 1996. 36

R. Cravotta. DSP directory 2003. http://www.reed-electronics.com/ednmag/contents/images/-286246.pdf, 2003. 100

R. Crisp. Direct Rambus technology: The next main memory standard. *IEEE Micro*, 17(6): 18–28, Nov./Dec. 1997. ISSN 0272-1732. 76

W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labont, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Supercomputing Conference 2003*, Nov. 2003. 95

C. A. Davatzikos, R. N. Bryan, and J. L. Prince. Image registration based on boundary mapping. *IEEE Trans. Medical Imaging*, 15, no. 1:112–115, 1996. 24

B. Davis, B. Jacob, and T. Mudge. The new DRAM interfaces: SDRAM, RDRAM and variants. *Lecture Notes in Computer Science*, 1940, 2000a. ISSN 0302-9743. 71

B. Davis, T. Mudge, B. Jacob, and V. Cuppu. DDR2 and low latency variants. In *Solving the Memory Wall Problem Workshop*, 2000b. 71

A. DeHon. The density advantage of configurable computing. *Computer*, 33(4):41–49, Apr. 2000. ISSN 0018-9162. 101

A. DeHon. Very large scale spatial computing. *Lecture Notes in Computer Science*, 2509, 2002. ISSN 0302-9743. 100, 156

D. Demigny, L. Kessal, R. Bourguiba, and N. Boudouani. How to use high speed reconfigurable FPGA for real time image processing? In *Fifth IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'00)*, page 240, 2000. 161

R. Desikan, S. W. Keckler, D. Burger, and T. Austin. Assessment of MRAM technology characteristics and architectures. Technical Report CS-TR-01-36, The University of Texas at Austin, Department of Computer Sciences, Apr 2001. 71

U. Diewald, T. Preusser, M. Rumpf, and R. Strzodka. Diffusion models and their accelerated solution in computer vision applications. *Acta Mathematica Universitatis Comenianae (AMUC)*, 70(1):15–31, 2001. 132

J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the International Conference on Supercomputing*, June 2002. 96

M. J. B. Duff. Thirty years of parallel image processing. In *Vector and Parallel Processing - VECPAR 2000*, volume 1981 / 2001, pages 419–438, 2000. 99

R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990. ISSN 0018-9162. 93

Elixent. http://www.elixent.com/. 102, 166

J. Ely. *Prospects for Using Variable Precision Interval Sotware in C++ for Solving Some Contemporary Scientific Problems*. PhD thesis, The Ohio State University, 1990. 30

B. Engquist and S. Osher. Stable and entropy-satisfying approximations for transonic flow calculations. *Math. Comp.*, 34(149):45–75, 1980. 55, 56

R. Enzler, C. Plessl, and M. Platzner. Virtualizing hardware with multi-context reconfigurable arrays. In *Field Programmable Logic and Application, 13th International Conference, FPL 2003*, pages 151–160, 2003. 101

Evans & Sutherland. Company's history. http://www.es.com/about_eands/history/index.asp, 2004. 116

R. Fernando and M. L. Kilgard. *The Cg Tutorial: The Definitive Guide To Programmable Real-Time Graphics*. Addison-Wesley, 2003. 116

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, C-21(9):948–960, 1972. 92

B. Fraguela, P. Feautrier, J. Renau, D. Padua, and J. Torrellas. Programming the FlexRAM parallel intelligent memory system. In *International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Jun. 2003. 96

F. Franchetti and M. Püschel. Short vector code generation and adaptation for DSP algorithms. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing; Conference Proceedings (ICASSP '03)*, 2003. 98

J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the Streams-C C-to-FPGA compiler: An applications perspective. In *9th ACM International Symposium on Field-Programmable Gate Arrays*, Feb. 2001. 158

I. Gavrichenkov. AMD Athlon 64 FX-51 vs. Intel Pentium 4 Extreme Edition 3.2GHz: clash of strong wills. http://www.xbitlabs.com/articles/cpu/display/athlon64-fx51_[8,9].html, 2003. 81

D. Geman, S. Geman, C. Graffigne, and P. Dong. Boundary detection by constrained optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 609–628, 1990. 21

D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991. ISSN 0360-0300. 32, 37

N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary-value problems using programmable graphics hardware. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2003. 133, 143

GPGPU. GPGPU - general purpose computation using graphics hardware. http://-www.gpgpu.org/. Mark J. Harris/Ed. 110, 142, 143, 181

Gray and Neuhoff. Quantization. *IEEETIT: IEEE Transactions on Information Theory*, 44(6): 2325–2383, 1998. 26

U. Grenander and M. I. Miller. Computational anatomy: An emerging discipline. *Quarterly Appl. Math.*, LVI, no. 4:617–694, 1998. 24

Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *Symp. on Field-Programmable gate Arrays (FPGA)*, Feb. 2004. 101, 181

C. Gwyn and P. Silverman. EUVL transition from research to commercialization. In *Photomask and Next-Generation Lithography Mask Technology X*, volume 5130, Apr. 2003. 104

J. Hammes, A. Bohm, C. Ross, M. Chawathe, B. Draper, and W. Najjar. High performance image processing on FPGAs. In *Los Alamos Computer Science Institute Symposium*, Oct. 2001. 101, 159

J. Hammes, R. Rinker, W. Böhm, and W. Najjar. Cameron: High level language compilation for reconfigurable systems. In *PACT 99,*, Okt. 1999. 158

M. Hanke and C. Groetsch. Nonstationary iterated Tikhonov regularization. *J. Optim. Theory and Applications*, 98:37–53, 1998. 24

M. J. Harris. Analysis of error in a CML diffusion operation. Technical report, UNC, 2002. 133

M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of Graphics Hardware 2002*, pages 109–118, 2002. 133

M. J. Harris, W. V. B. III, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of Graphics Hardware 2003*, 2003. 133

R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Design, Automation and Test in Europe - DATE 2001*, Mar. 2001. 94, 102, 165

R. Hartenstein. Data-stream-based computing: Models and architectural resources. In *International Conference on Microelectronics, Devices and Materials (MIDEM 2003)*, Ptuj, Slovenia, Oct. 2003. 84, 100

H.Becker, S.Kilian, and S.Turek. Some concepts of the software package feast. In J. et al., editor, *Proc: Vector and Parallel Processing - VECPAR98*, pages 271–284, 1999. 26

S. Henn and K. Witsch. Iterative multigrid regularization techniques for image matching. *SIAM J. Sci. Comput. (SISC)*, Vol. 23 no. 4:pp. 1077–1093, 2001. 24

M. Herz, R. Hartenstein, M. Miranda, E. Brockmeyer, and F. Catthoor. Memory organisation for stream-based reconfigurable computing. In *9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002*, Dubrovnik, Croatia, Sep. 2002. 84

G. A. Hewer, C. Kenney, and B. S. Manjunathg. Variational image segmentation using boundary functions. *IEEE Transactions on Image Processing*, 7(9), 1998. 21

N. J. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993. 30

N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. ISBN 0-89871-521-0. 26, 29

M. Hopf and T. Ertl. Accelerating 3D convolution using graphics hardware. In *Proc. Visualization '99*, pages 471–474. IEEE, 1999. 132

M. Hopf and T. Ertl. Accelerating Morphological Analysis with Graphics Hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*, pages 337–345, 2000a. 132

M. Hopf and T. Ertl. Hardware Accelerated Wavelet Transformations. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '00*, pages 93–103, 2000b. 132

Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: An efficient approach to predicting and optimizing memory behavior. In *International Symposium on Computer Architecture (ISCA) 2002*, 2002. 83

IEC. *Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics*, second edition edition, Nov. 2000. 7

Intel. Intel Itanium 2 Processor. http://www.intel.com/products/server/processors/server/-itanium2/, 2004a. 98

Intel. Intel Pentium 4 Processor. http://www.intel.com/products/desktop/processors/-pentium4/, 2004b. 79, 81

IPflex. http://www.ipflex.com/english/. 102, 166

L. Jackson. Roundoff noise analysis for fixed-point digital filters realized in cascaded or parallel form. *IEEE Trans. Audio Electroacoust.*, AU-18:107–122, 1970. 35

U. J. Kapasi, W. J. Dally, B. Khailany, J. D. Owens, and S. Rixner. The Imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 282–288, Sep. 2002. 95

U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, Aug. 2003. 95

T. Kapur, W. Grimsol, W. WellsIII, and R. Kikinis. Segmentation of brain tissue from magnetic resonance image. *Medical Image Analysis*, 1(2), 1997.

T. Karkhanis and J. Smith. A day in the life of a cache miss. In *International Symposium on Computer Architecture (ISCA) 2002*, 2002. 82

J. Kačur and K. Mikula. Solution of nonlinear diffusion appearing in image smoothing and edge detection. *Appl. Numer. Math.*, 17 (1):47–59, 1995. 43

B. Kawohl and N. Kutev. Maximum and comparison principle for one-dimensional anisotropic diffusion. *Math. Ann.*, 311 (1):107–123, 1998. 19

H. Keding, F. Hürtgen, M. Willems, and M. Coors. Transformation of floating-point into fixed-point algorithms by interpolation applying a statistical approach. In *International Conference on Signal Processing Applications & Technology 1998 (ICSPAT-98)*, 1998. 36

B. Keeth and R. J. Baker. *DRAM Circuit Design : A Tutorial*. Wiley-IEEE Press, Nov 2000. 71

B. Khailany, W. Dally, S. Rixner, U. Kapasi, J. Owens, and B. Towles. Exploring the VLSI scalability of stream processors. In *International Conference on High Performance Computer Architecture (HPCA-2003)*, 2003. 95, 168

S. Kichenassamy. The perona-malik paradox. *SIAM J. Appl. Math.*, 57:1343–1372, 1997. 19

S. Kim, K.-I. Kum, and W. Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems*, 45(11), 1998. 36

T. Kim and M. Lin. Visual simulation of ice crystal growth. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2003. 133

S. Klupsch. Design, integration and validation of heterogeneous systems. In *2nd IEEE International Symposium on Quality Electronic Design (ISQED 2001)*, Mar. 2001. 158

S. Klupsch, M. Ernst, S. A. Huss, M. Rumpf, and R. Strzodka. Real time image processing based on reconfigurable hardware acceleration. In *Proc. Heterogeneous reconfigurable Systems on Chip*, 2002. 156

P. Kosmol. *Optimierung und Approximation*. de Gruyter Lehrbuch, 1991. 60

J. Krueger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003. ISSN 0730-0301. 133

A. Lefohn, J. Kniss, C. Handen, and R. Whitaker. Interactive visualization and deformation of level set surfaces using graphics hardware. In *Proc. Visualization*, pages 73–82. IEEE CS Press, 2003. 127, 131

A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive deformation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 2004. 128

G. Lenerz. Silicon Graphics history. http://sgistuff.g-lenerz.de/hardware/timeline.html, 2004. 116

C. Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*. Wiley, 2000. 99

E. Loh and G. W. Walster. Rump's example revisited. *Reliab. Comput.*, 8(3):245–248, 2002. ISSN 1385-3139. 29

F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens. Multi–modal volume registration by maximization of mutual information. *IEEE Trans. Medical Imaging*, 16, no. 7:187–198, 1997. 24

K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *27th Annual International Symposium on Computer Architecture (27th ISCA-2000) Computer Architecture News*, Vancouver, British Columbia, Canada, June 2000. ACM SIGARCH / IEEE. 92, 95, 103

R. Malladi, J. A. Sethian, and B. C. Vemuri. Shape modelling with front propagation. *IEEE Trans. Pattern Anal. Machine Intell.*, 17, 1995. 21

DirectX: multimedia application programming interfaces. Microsoft, http://-www.microsoft.com/windows/directx/default.aspx. 111

Microsoft. Longhorn Developer Center. http://msdn.microsoft.com/longhorn. 119

DirectX9 Programmable HLSL Shaders. Microsoft, http://msdn.microsoft.com/library/-default.asp?url=/nhp/default.asp?contentid=28000410, 2003. 118

G. Moore. No exponential is forever ... but we can delay "forever". http://www.intel.com/-research/silicon/mooreslaw.htm, Feb. 2003. Event: International Solid State Circuits Conference (ISSCC). 103

G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965. 71

MPI committee. The message passing interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/, 2004. 99

L. P. Mulcahy. Two's-complement fixed-point multiplications errors. Technical Report AD-A086826, Naval Ocean System Center, San Diego, CA, 1980. 35

D. Mumford and J. Shah. Boundary detection by minimizing functionals. In *Proceedings, CVPR '85 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition)*, IEEE Publ. 85CH2145-1., pages 22–26, 1985. 21

R. Murphy and P. M. Kogge. The characterization of data intensive memory workloads on distributed PIM systems. In *Intelligent Memory Systems Workshop*, Boston, MA, Nov. 2000. 96

Nallatech. http://www.nallatech.com/. 99

NVIDIA. Cg programming language. http://developer.nvidia.com/page/cg_main, 2002. 118, 126, 144

NVIDIA. GeForce FX. http://www.nvidia.com/page/fx_desktop.html, 2004. 81

OpenGL: graphics application programming interface. OpenGL Architectural Review Board (ARB), http://www.opengl.org/. 111

GLSL - OpenGL Shading Language. OpenGL Architectural Review Board (ARB), http://-www.opengl.org/documentation/oglsl.html, 2004. 118, 126

S. J. Osher and J. A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on Hamilton–Jacobi formulations. *J. of Comp. Physics*, 79:12–49, 1988. 21, 55

PACT. http://www.pactcorp.com/. 102, 166, 167, 168, 175

P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12:629–639, 1990. 19

picoChip. http://www.picochip.com/. 102, 166

T. Preußer and M. Rumpf. An adaptive finite element method for large scale image processing. *Journal of Visual Comm. and Image Repres.*, 11:183–195, 2000. 44

D. M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California, Berkeley, CA, USA, 1992. 29

QuickSilver. http://www.quicksilvertech.com/. 102, 166

P. Ranganathan, S. V. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *ISCA*, pages 124–135, 1999. 98

D. Risley. A CPU history. http://www.pcmech.com/show/processors/35/, 2001. 79

S. Rixner. *Stream Processor Architecture*. Kluwer Academic Publishers, 2002. 95

T. Rolfe. Distributed multiprocessor environments. *J. Comput. Small Coll.*, 18(2):95–104, 2002. 93

S. M. Rump. *Algorithms for verified inclusions in theory and practice*, pages 109–126. Academic Press Professional, Inc., 1988. ISBN 0-12-505630-3. 29

M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Proceedings ICIP'01*, volume 3, pages 1103–1106, 2001a. 127

M. Rumpf and R. Strzodka. Nonlinear diffusion in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*, pages 75–84. Springer, 2001b. 132, 139

M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings VIIP'01*, pages 193–202, 2001c. 132

M. Rumpf and R. Strzodka. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Graphics Processor Units: New Prospects for Parallel Computing. Springer, 2005. 110

K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA 2003*, pages 422–433, 2003. 92, 95

Y. Sato, S. Nakajima, N. Shiraga, H. Atsumi, S. Yoshida, T. Koller, G. Gerig, and R. Kikinis. Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images. *Medical Image Analysis*, 2(2), 1998.

M. J. Schulte and E. E. Swartzlander. Software and hardware techniques for accurate, self-validating arithmetic. In R. B. Kearfott and V. Kreinovich, editors, *Applications of interval computations: Papers presented at an international workshop in El Paso, Texas, February 23–25, 1995*, volume 3 of *Applied optimization*, pages 381–404, Norwell, MA, USA, and Dordrecht, The Netherlands, 1996. Kluwer Academic Publishers Group. ISBN 0-7923-3847-2. 39

SEMATECH. International technology roadmap for semiconductors (ITRS). http://-public.itrs.net/Files/2003ITRS/Home2003.htm, 2003. 71, 103

J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999. 20, 21, 130

A. K. Sharma. *Advanced Semiconductor Memories : Architectures, Designs, and Applications*. Wiley-IEEE Press, Oct 2002a. 71

A. K. Sharma. *Semiconductor Memories : Technology, Testing, and Reliability*. Wiley-IEEE Press, Aug 2002b. 71

K. Siddiqi, Y. B. Lauzière, A. Tannenbaum, and S. W. Zucker. Area and length minimizing flows for shape segmentation. *IEEE Transactions on Image Processing*, 7(3), 1998.

Silicon Software. *microEnable Users Guide*, 1999. 156

T. Stansfield. Wordlength as an architectural parameter for reconfigurable computing devices.

Field-Programmable Logic and Applications, Proceedings Lecture Notes in Computer Science, 2438:667–676, 2002. 102

Star Bridge Systems Inc. http://www.starbridgesystems.com/. 99

T. L. Sterling and H. P. Zima. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *SC'2002 Conference CD*, Baltimore, MD, Nov. 2002. IEEE/ACM SIGARCH. pap105. 96

R. Strzodka. Virtual 16 bit precise operations on RGBA8 textures. In *Proceedings VMV'02*, pages 171–178, 2002. 121, 141

R. Strzodka, M. Droske, and M. Rumpf. Fast image registration in DX9 graphics hardware. *Journal of Medical Informatics and Technologies*, 6:43–49, Nov 2003. 143

R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing*, 2004. to appear. 143

B. Su, E.-W. Hu, J. Manzano, S. Regula, J. Wang, and L. W. Leung. A new source-level benchmarking for DSP processors. In *Proceedings of Global Signal Processing Expo & Conference (GSPx) 2004*, 2004. 100

A. Sud, D. Manocha, N. K. Govindaraju, and S. eui Yoon. Parallel occlusion culling for interactive walkthroughs using multiple GPUs. Technical report, UNC Computer Science, 2002. 99

T. Sueyoshi and M. Iida. Configurable and reconfigurable computing for digital signal processing. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E85A(3):591–599, 2002. 102, 176, 181

J. Suh, E.-G. Kim, S. P. Crago, L. Srinivasan, and M. C. French. A performance analysis of PIM, stream processing, and tiled processing on memory-intensive signal processing kernels. In D. DeGroot, editor, *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-03)*, volume 31, 2 of *Computer Architecture News*, pages 410–421, New York, June 9–11 2003. ACM Press. 103, 181

W. Sung and K. Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Transactions on Signal Processing*, 43(12), 1995. 36

M. B. Taylor, J. S. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002. 95

R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI Signal Processing*, 28(1):7–27, June 2001. 100, 161, 181

J. P. Thirion. Image matching as a diffusion process: An analogy with Maxwell's demon. *Medical Imag. Analysis*, 2:243–260, 1998. 24

V. Thomée. *Galerkin - Finite Element Methods for Parabolic Problems*. Springer, 1984. 44

H. Tian, S. Lam, and T. Srikanthan. Area-time efficient between-class variance module for adaptive segmentation process. *IEEE Proceedings: Vision, Image and Signal Processing*, 150(4):263–269, 2003. 161

TOP500 committee. TOP500 supercomputer sites. http://www.top500.org/. 99

B. Ulmann and R. Hoffmann. Instruction stream processing beyond vector computing. In *Proceedings of the 2002 conference on Massively-Parallel Computing Systems*, pages 118–123, 2002. 95

M. Urabe. Roundoff Error Distribution in Fixed-Point Multiplication and A Remark About the Rounding Rule. *SIAM Journal of Numerical Analysis*, 5(2):202–210, June 1968. 35

I. Vladimirov and P. Diamond. A uniform white-noise model for fixed-point roundoff error in digital systems. *Automation and Remote Control*, 63(5):753–765, 2002. 35

S. A. Wadekar and A. C. Parker. Accuracy sensitive word-length selection for algorithm optimization. In *International Conference on Computer Design (ICCAD'98)*, pages 54–61, 1998. 36

J. Waldemark, M. Millberg, T. Lindblad, and K. Waldemark. Image analysis for airborne reconnaissance and missile applications. *Pattern Recognition Letters*, 21(3):239–251, Mar. 2000. 161

J. Weickert. *Anisotropic diffusion in image processing*. Teubner, 1998. 17, 19, 43, 46, 49

J. Weickert, K. Zuiderveld, B. ter Haar Romeny, and W. Niessen. Parallel implementations of AOS schemes: A fast way of nonlinear diffusion filtering. In *Proc. Fourth IEEE International Conference on Image Processing*, volume 3, pages 396–399, Santa Barbara, CA, Oct 1997. 44

P. Welch. A fixed-point fast Fourier transform error analysis. *IEEE Trans. Audio Electroacoust.*, AU-17:151–157, 1969. 35

R. Westermann. The rendering of unstructured grids revisited. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '01*. Springer, 2001. 26

M. Wilkes. The memory gap (keynote). In *Solving the Memory Wall Problem Workshop*, 2000. http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf. 71

M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr. System level fixed-point design based on an interpolative approach. In *Design Automation Conference 1997 (DAC-97)*, 1997a. 36

M. Willems, V. Bürsgens, and H. Meyr. Fridge: Floating-point programming of fixed-point digital signal processors. In *Int. Conf. On Signal Processing Applications & Technology 1997 (ICSPAT-97)*, 1997b. 36

K. Williston, M. Tsai, and J. Bier. DSP benchmark results for the latest processors. In *Proceedings of Global Signal Processing Expo & Conference (GSPx) 2004*, 2004. 100

P. W. Wong. Quantization and roundoff noises in fixed-point FIR digital filters. *IEEE Transactions on Signal Processing*, 39(7):1552–1563, 1991. 35

P. W. Wong. Quantization noise, fixed-point multiplicative roundoff noise, and dithering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 38(2):286–300, l99O. 35, 49

Xilinx Inc. http://www.xilinx.com. 156, 157

C. Xu and J. L. Prince. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing*, 7(3), 1998. 21

Y.-L. You, W. Xu, A. Tannenbaum, and M. Kaveh. Behavoiral analysis of anisotropic diffusion in image processing. *IEEE Trans. Image Proc.*, 5:1539–1553, 1996. 19

Acronyms

AGP	Accelerated Graphics Port
ALU	Arithmetic and Logic Unit
ANSI	American National Standards Institute
API	Application Programming Interface
ARB	Architectural Review Board
ASIC	Application Specific Integrated Circuit
ASSP	Application Specific Standard Product
BREG	Backward Register
BL	burst length
CFL	Courant-Friedrichs-Levy condition
CLB	Configurable Logic Block
CM	Configuration Manager
CPU	Central Processor Unit
CSoC	Configurable System-on-a-Chip
DDR	Double Data Rate (memory)
DDR1	First Generation DDR memory
DDR2	Second Generation DDR memory
DR	Data Rate
DSB	data-stream-based
DSM	Distributed Shared Memory
DSP	Digital Signal Processing
DRAM	Dynamic RAM
EUV	Extreme Ultra-Violet
FE	Finite Element

FIFO	First In First Out
FLOPS	floating point OPS
FP	Fragment Processor
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FREG	Forward Register
FSB	Front Side Bus
GDDR	Graphics DDR memory
GDDR3	Third Generation GDDR memory
GPU	Graphics Processor Unit
HDL	Hardware Description Language
HLL	High Level Language
HPC	High Performance Computing
IEEE	Institute of Electrical and Electronics Engineers
IEC	International Electrotechnical Commission
IO	Input/Output
IP	Intellectual Property
ITRS	International Technology Roadmap for Semiconductors
ISB	instruction-stream-based
LUT	lookup table
MAC	multiply-accumulate instruction
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MPP	Massively Parallel Processing
NML	Native Mapping Language
NUMA	Non-Uniform Memory Access
OPS	operations per second
PAE	Processing Array Element

PBO	Pixel Buffer Object
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PDE	partial differential equation
PE	processing element
PIM	Processor-in-Memory
PS	Pixel Shader
QDR	Quad Data Rate (memory)
RAM	Random Access Memory
RC	Reconfigurable Computing
RL	Reconfigurable Logic
SDR	Single Data Rate (memory)
SDRAM	Synchronous DRAM
SI	Système International d'Unités
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMT	Simultaneous Multi-Threading
SoC	System-on-a-Chip
SPMD	Single Program Multiple Data
SRAM	Static RAM
SSE	Streaming SIMD Extensions
tCLK	clock cycle time
tCL	column access latency
tCAS	column active time
tCP	column precharge time
TMU	texture mapping unit
tRAS	row active time
tRCD	row to column delay

tRP	row precharge time
VLIW	Very Long Instruction Word
VBO	Vertex Buffer Object
VP	Vertex Processor
VS	Vertex Shader
WGF	Windows Graphics Foundation
XDR	eXtreme Data Rate (memory)
XPP	eXtreme Processing Platform

Index