Diss. ETH No. 18596

Accelerated Nonrigid Image Registration

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Sciences ETH Zurich

presented by
JONATHAN ROHRER
M.Sc. ETH in Electrical Engineering and Information Technology
born 16 July 1980
citizen of Sachseln OW

accepted on the recommendation of
Prof. Dr. Gábor Székely, examiner
Prof. Dr. Gerhard Tröster, co-examiner
Dr. Ton Engbersen, co-examiner

2009

Abstract

Many tasks in medical image analysis require the fusion of two images, for example to combine information provided by different image acquisition devices or monitor disease progression over time. Image registration is the process of aligning two images, such that corresponding points can be related. For this purpose, one image is deformed to match the other one. Rigid registration techniques try to achieve an alignment by scaling, rotating and translating one of the images. Rigid registration is only adequate for special cases because usually the anatomical structures in the images are not rigid and therefore more complex nonrigid transformation models are required. A problem with nonrigid registration methods is their high computational cost yielding registration times in the order of hours for typical 3D images. While this is inconvenient for some applications, it is prohibitive for others, especially in the context of intraoperative scenarios.

This dissertation presents a nonrigid registration algorithm and implementation achieving sub-minute runtimes on a system based on two Cell/B.E. processors. This processor contains multiple processor cores on one chip and is designed for computationally intensive workloads. A speedup of more than 2500× was achieved compared to a sequential open-source implementation running on a general-purpose processor. Although optimized for the Cell/B.E. architecture, the presented algorithm also achieves a high efficiency on more recent general-purpose multicore architectures. A scalability analysis shows that the algorithm has the potential to exploit future architectures with more cores.

The algorithm bases on the B-spline transformation model, which has been applied successfully to a wide range of nonrigid registration problems. Furthermore, it uses mutual information, which is probably the most common similarity metric for multimodal image registration. Mutual information allows registration of images which were obtained from different acquisition devices.

Zusammenfassung

Für viele Anwendungen im Bereich der medizinischen Bildanalyse müssen zwei Bilder zusammengeführt werden, zum Beispiel um Information aus zwei verschiedenen bildgebenden Verfahren zu kombinieren oder um die Entwicklung einer Krankheit über eine gewisse Zeitspanne zu überwachen. Bildregistrierung versucht eines der Bilder so zu deformieren, dass entsprechende Punkte zusammenfallen, wenn die Bilder übereinander gelegt werden. Rigide Registrierungs-Techniken versuchen mittels Rotation, Translation und Skalierung eine Übereinstimmung zu erreichen. Dies ist nur in Spezialfällen möglich, da die meisten Strukturen in den Bildern aus weichen Geweben bestehen. Darum werden im Allgemeinen komplexere Modelle benötigt. Ein Problem sogenannter elastischer Methoden ist deren hoher Rechenaufwand, der für typische 3D Bilder zu Registrierungszeiten von mehreren Stunden führen kann. Für einige Anwendungen ist dies störend, andere verunmöglicht es, vor allem im intraoperativen Bereich.

Diese Dissertation präsentiert einen elastischen Registrierungsalgorithmus, welcher für die Registrierung auf einem System mit zwei Cell/B.E. Prozessoren weniger als eine Minute benötigt. Dieser Prozessor vereinigt mehrere Prozessorkerne auf einem Chip und ist für rechenintensive Anwendungen ausgelegt. Im Vergleich zu einer sequenziellen open-source Implementierung wurde eine mehr als 2500-fache Beschleunigung erzielt. Obwohl der präsentierte Algorithmus für die Cell/B.E. Architektur optimiert wurde, erreicht er auch auf Standard-Mehrkernprozessoren eine hohe Effizienz. Eine Skalierbarkeitsanalyse zeigt, dass auch das Potential von Prozessoren mit mehr Kernen ausgenützt werden kann.

Der Algorithmus basiert auf dem B-spline Transformationsmodell, welches schon erfolgreich auf verschiedenste elastische Registrierungsprobleme angewandt wurde. Die Ähnlichkeit wird mit der Transinformation gemessen, dem wahrscheinlich verbreitetsten Ähnlichkeitsmass für Bilder von verschiedenen bildgebenden Verfahren.

Contents

Abstract iii

Zusammenfassung v

1 Introduction 1
- 1.1 Image Registration 1
- 1.2 Applications 2
- 1.3 Classification 3
- 1.4 Organization 4

2 Nonrigid Image Registration 5
- 2.1 Similarity Metrics 5
- 2.2 Physics-Based Approaches 8
- 2.3 Demons Algorithm 10
- 2.4 B-Spline Model 11
- 2.5 Piecewise Rigid Registration 12

3 Accelerating Nonrigid Image Registration 13
- 3.1 Overview 13
- 3.2 Multiprocessors 14
  - 3.2.1 Shared-Memory Architectures 14
  - 3.2.2 Message-Passing Architectures 17
  - 3.2.3 Programming Models 17
  - 3.2.4 Early Nonrigid Registration Work 20
  - 3.2.5 Biomechanical Models 20
  - 3.2.6 Demons Algorithm 22
  - 3.2.7 B-spline Algorithms 22
  - 3.2.8 Conclusions 23
- 3.3 Trends in Processor Development 24

3.3.1 Instruction-Level Parallelism 24
3.3.2 Multicore Processors 26
3.3.3 Asymmetric Processors 26
3.3.4 Accelerators 27
3.3.5 Memory 27
3.3.6 Programming 28
3.4 Cell/B.E. 29
3.4.1 Overview 29
3.4.2 Processor Architecture 29
3.4.3 PPE 29
3.4.4 SPE 30
3.4.5 On-Chip Communication 32
3.4.6 System Architecture 32
3.4.7 Programming Models 34
3.4.8 Rigid Registration 36
3.5 GPU 37
3.5.1 Overview 37
3.5.2 System Architecture 38
3.5.3 Processor Architecture 38
3.5.4 Programming Model 40
3.5.5 Early Nonrigid Registration Work 41
3.5.6 Gradient Flow Algorithms 41
3.5.7 Demons Algorithm 42
3.5.8 Finite Element Algorithm 44
3.5.9 Multimodal Registration 44
3.5.10 Conclusions 45
3.6 FPGA 46
3.6.1 Architecture 46
3.6.2 Volume-Subdivision Algorithm 46
3.7 Conclusions 47

4 Implemented Registration Algorithm 49
4.1 Basic Features 49
4.2 B-Spline Transformation Model 50
4.3 Mutual Information 52
4.4 Moving Image Model 54
4.5 Optimization Strategy 55

5 Parallel Algorithm Design 59
5.1 Problem Analysis . . . 59
5.1.1 The Algorithm . . . 59
5.1.2 Performance Bottlenecks . . . 60
5.2 Optimizations . . . 61
5.2.1 B-Spline Interpolation . . . 61
5.2.2 Use of Lookup Tables . . . 62
5.2.3 Vectorized B-Spline Interpolation . . . 62
5.2.4 Optimized Transformation Field Evaluation . . . 64
5.2.5 Optimized Image Interpolation . . . 64
5.2.6 Gradient Calculation . . . 66
5.2.7 Conclusions . . . 69
5.3 Parallelization Platform . . . 69
5.3.1 Profiling . . . 69
5.3.2 Theoretical Limitations . . . 70
5.3.3 Limitations for GPUs . . . 70
5.3.4 Target Platform for Parallelization . . . 71
5.3.5 Master-Worker Programming Model . . . 72
5.3.6 Conclusions . . . 72
5.4 Data Locality . . . 73
5.4.1 Producer-Consumer Locality . . . 73
5.4.2 Spatial Locality . . . 75
5.4.3 Storing Intermediate Results for Reuse . . . 77
5.5 Parallelism 1: Multiple SPEs . . . 78
5.5.1 Parallelization . . . 78
5.5.2 Compute Kernels . . . 78
5.5.3 Synchronization and Coherency . . . 79
5.5.4 Communication . . . 80
5.5.5 Model Overview . . . 80
5.6 Data Transfers . . . 81
5.6.1 Double Buffering . . . 81
5.6.2 Caching of Data . . . 82
5.6.3 Probabilistic Prefetching of Moving Image Data . . . 83
5.6.4 Data Replication . . . 84
5.7 Parallelism 2: Multiple Cell/B.E. Processors . . . 85
5.7.1 The Parallelized Phases . . . 85
5.7.2 The Communication Phases . . . 86
5.7.3 The Sequential Phases . . . 88

5.7.4 Conclusions 88

6 Performance Analysis 91

6.1 Benchmark 92
6.1.1 Setup 92
6.1.2 Results Overview 93
6.1.3 Code and Data Structures 94
6.1.4 Re-use of the Transformed Image and its Gradient . . 95
6.1.5 Grouping 96
6.1.6 Manual Vectorization 98
6.1.7 Vector Alignment 99
6.1.8 Double Buffering 100
6.1.9 Parallelism 100
6.1.10 Conclusions 100
6.2 Modeling the Runtime 101
6.2.1 Measuring Runtime 102
6.2.2 Parallelized Functions 103
6.2.3 Communication Overhead 104
6.2.4 The Sequential Part 106
6.3 Scalability 1: Many-Core 109
6.3.1 Fixed Problem Size 109
6.3.2 Reducing Gradient Collection Overhead 110
6.3.3 Reducing Histogram Collection Overhead 114
6.3.4 Scaling Problem Size 119
6.3.5 Conclusions 121
6.4 Memory Considerations 122
6.4.1 The Memory Wall 122
6.4.2 Bandwidth Requirements 122
6.4.3 Prefetching 124
6.4.4 Cache Miss Penalty 126
6.4.5 Conclusions 128
6.5 Scalability 2: Cell/B.E. Cluster 129
6.5.1 Communication Cost on a QS21 Cluster 129
6.5.2 Extending the Model 133
6.5.3 Measurements on the QS21 Cluster 136

7 Experiments 137
7.1 Simulated MR Images 137
7.2 CT Images 140
7.3 Comparison with Other Algorithms 141

8 Conclusions and Outlook 143

A List of abbreviations 147

Curriculum Vitae 161

List of Figures

2.1 Simulated MR images of the brain 6
2.2 Joint histograms 7

3.1 Cache coherency problem 15
3.2 Thread synchronization problem 16
3.3 Cell/B.E. processor 30
3.4 Cell/B.E. blade 33
3.5 Porting applications to the Cell/B.E. architecture 34
3.6 Cell/B.E. programming models 35
3.7 System architecture with GPU. 38
3.8 Graphics pipeline 39
3.9 Speedup comparison 47

4.1 2D cubic B-spline base function 51
4.2 B-spline transformation model 52

5.1 Producer-consumer diagram 61
5.2 Vectorized B-spline interpolation 63
5.3 Optimized image interpolation 65
5.4 Producer-consumer diagram after pipelining 73
5.5 Mapping images to memory 76
5.6 Producer-consumer model with temporary buffering 77
5.7 Parallelization scheme 81
5.8 Producer-consumer diagram with double buffering 82
5.9 Probabilistic prefetching 83
5.10 Moving image data replication 85

6.1 Speedup 94
6.2 Rigid transformation with varying block width 97
6.3 Nonrigid transformation with varying block width 98

6.4 Model of t_{B1} 103
6.5 Model of t_{B2} 103
6.6 Model of t_{hsync} 105
6.7 Model of t_{gsync} 105
6.8 Model of t_{mi} 107
6.9 Model of t_{gd} 108
6.10 Scalability for fixed problem size 109
6.11 Efficiency for fixed problem size 110
6.12 Parallel reduce operation: gradient 112
6.13 The reduce operation for large arrays is memory bound . . . 112
6.14 Scalability with improved gradient synchronization 115
6.15 Collective communication of SPEs 116
6.16 Collective reduce operation: histogram 117
6.17 Parallel reduce operation: histogram 118
6.18 Scalability with improved synchronization 119
6.19 Efficiency with improved synchronization 120
6.20 Isoefficiency curve 120
6.21 Bandwidth requirements: phase B1 122
6.22 Bandwidth requirements: phase B2 123
6.23 Memory traffic depending on prefetch threshold 125
6.24 Block fetches depending on prefetch threshold 125
6.25 Duration of a DMA transfer 127
6.26 Duration of a reduce operation on the QS21 cluster 129
6.27 Bandwidth of a reduce operation on the QS21 cluster 130
6.28 Start-up time on the QS21 cluster 131
6.29 Asymptotic bandwidth on the QS21 cluster 132
6.30 Model of the runtime of a reduce operation 133
6.31 Scalability on the cluster: One processor per node 134
6.32 Efficiency on the cluster: One processor per node 135
6.33 Measured runtime on the cluster 135

7.1 Deformed BrainWeb images 138
7.2 Registered BrainWeb images 140
7.3 Rigid and nonrigid registration of abdominal CT images . . 141

List of Tables

5.1 Algorithm profiling 69

6.1 Runtime on different platforms 93
6.2 Parameters of the runtime model 101
6.3 Phases of the algorithm 102

7.1 Registration runtime: BrainWeb images 139
7.2 Registration runtime: abdominal CT images 142

1

Introduction

1.1 Image Registration

Assume that we have two images, a fixed[1] image and a moving[2] image. These images show the same or a similar "scene" but are not identical. Image registration is the process of bringing the two images into geometric alignment by finding the corresponding points in the two images. Possible reasons for a misalignment are

- The images were acquired from different viewpoints.
- The images are from different acquisition devices. This case is very common in medical imaging, where different modalities like CT (computer tomography), MRI (magnetic resonance imaging) and ultrasound are used. Solving such problems is called multimodal or intermodal registration.
- The images were acquired at different times and certain objects in the image may have changed their position or shape.

---

[1] The fixed image sometimes is also called static, target, reference, study or baseline image.

[2] Other names for the moving image are source, floating, template, or repeat image.

- The image of a scene is registered to a model of the scene.

The output of the registration is a transformation function $\mathcal{T}$, which maps points x_{fix} in the fixed image to their corresponding points $x_{mov} = \mathcal{T}(x_{fix})$ in the moving image.

There are various articles reviewing registration techniques and their applications in general, for example by Brown [15] or Zitová and Flusser [109]. This dissertation is about medical image registration. In the next section, a small number of example applications will be discussed.

1.2 Applications

An early motivation behind medical image registration was the desire to fuse information about one patient obtained from different imaging modalities. It became common for patients to be imaged with different tomographic modalities during treatment planning or diagnosis. Image registration compensated for differences in image resolution and the position of the patient in the image.

Another application arises when patients are imaged multiple times, for example for monitoring the growth of a tumor. Usually it is not possible to reposition the patient perfectly on subsequent visits. Image registration enables alignment of such images. Furthermore, many structures of the human body change their shape over time. Therefore, the images may also be affected by soft-tissue deformations, for example because of respiratory motion.

In a third possible application, image registration can establish correspondence between a patient image and an atlas image. The atlas image can be a model or the image of a typical subject. It is annotated, meaning that a segmentation of anatomical structures in the atlas is available. Detecting and labeling these structures is a time consuming and subjective task. A successful registration enables an automatic transfer of the labels from the atlas to the patient image, such that only one or a small number of atlases need to be segmented manually.

Furthermore, it is possible to acquire images in the operating room during surgery, which also opens new applications of image registration. These images can provide guidance to the surgeon, for example by providing improved contrast between healthy and diseased tissue. They also expose information about tissue below the surface. However, the constraints of an operating room allow less flexibility in the choice of the imaging modality than diagnostic acquisitions. With the goal of augmenting the information provided to the surgeon, image registration allows fusion of intraoperative with preoperative images of higher spatial resolution, higher signal to noise ratio or different modalities.

1.3 Classification

The large number of applications of image registration and their diversity has triggered the emergence of many registration methods. These can be classified based on their choices for four different properties [15]: feature space, search space, search strategy, and similarity metric.

The chosen feature space defines what information is extracted from the images. This can simply be the image intensity at a given pixel (or voxel in 3D), but other choices are possible. For example, the images can be transformed using Fourier or wavelet transformations. Salient points or objects, such as edges, surfaces, corners, and line intersections, can be extracted manually or preferably automatically.

The search space is determined by the model for the transformation function $\mathcal{T}$. Its choice usually depends on the expected distortions. A distinction is made between rigid and nonrigid[3] mapping. For 3D images, rigid or affine transformations are expressed by between six (three translations and three rotations) and twelve (additionally three scalings and possibly three shears) parameters. These are global transformations, which means that the same parameters apply to the entire image. Nonrigid transformations can also be local (each point is allowed to move freely) or semi-local (based on a number of parameters with local influence).

A search strategy has to be chosen to find the parameters of the transformation model yielding the “best” mapping between fixed and moving image. In many cases, a multidimensional optimization problem needs to be solved. Standard optimization techniques, such as Powell’s method or gradient descent, can be applied.

A similarity metric is required to judge if a certain mapping function is better than another one. The choice typically depends on the modalities of the two images. The simplest similarity metric is the sum of squared differences, which usually can be applied in the case of intramodal registration,

---

[3] Other terms used for nonrigid are nonlinear, elastic, deformable, and curved

where one assumes that corresponding points have the same intensity. However, usually more complex metrics are required in the case of intermodal registration.

1.4 Organization

Spatial alignment of clinical images can seldomly be approximated with rigid transformations and much of the most challenging work today is in the field of nonrigid registration. Such methods face two major obstacles for their widespread adoption in clinical environments: computational cost and difficulty in validating the results [25].

This dissertation addresses the computational efficiency of nonrigid registration. Workstation PCs (personal computers) typically require hours to (nonrigidly) register two images, which is not acceptable for many applications. Most of the faster solutions are restricted to expensive supercomputing systems, which limits their availability.

First, nonrigid registration methods are briefly discussed in chapter 2. In chapter 3, trends in processor development and emerging architectures are discussed along with accelerated nonrigid registration algorithms running on these platforms.

The main contribution of this dissertation is a multimodal nonrigid registration method which achieves runtimes below a minute on relatively low-cost platforms. The implemented algorithm is described in chapter 4. Its optimization and parallelization is discussed in chapter 5.

In chapter 6, performance and scalability of the algorithm are addressed. It includes an analysis of the consequences of the trends in processor development. Experimental results follow in chapter 7.

2

Nonrigid Image Registration

There are many articles summarizing image registration in general [15, 109], medical image registration [41, 65], mutual-information based image registration [81] and nonrigid image registration [25]. In this chapter, the most important similarity metrics and transformation models are briefly described.

2.1 Similarity Metrics

Similarity metrics are criteria to judge the similarity of two images. These metrics are usually calculated directly or indirectly from the gray-scale intensity values of the images. Such intensity-based approaches assume that the correct registration yields the highest similarity of the fixed and the transformed moving image. Geometric approaches on the other hand rely on the identification of corresponding anatomical elements in both images, typically surfaces, curves or point landmarks. This step can either be manual or automatic. The correspondence for the rest of the image is established by interpolation.

Geometric and intensity approaches may also be combined, for example by providing a robust initial transformation based on landmarks and refining it with intensity-based methods. Some of the most frequently used similarity metrics are briefly presented throughout the rest of this section. A more detailed discussion can be found in [41].

Sum of squared differences (SSD) is one of the simplest similarity metrics. The square of the difference of the fixed image intensity and the respective moving image intensity is accumulated over all fixed image voxels. The registered images are assumed to be identical, except for Gaussian noise. This means that SSD is not suitable for multimodal registration, where this assumption usually is violated. Different modalities usually result in different intensities for one specific structure. There are also intramodal scenarios where SSD is not an optimal choice, for example if contrast agents are used in one image to highlight certain structures.

The correlation coefficient is an intramodal similarity metric making less strict assumptions than SSD. It is optimal if there is a linear relationship between corresponding image intensities. It can be seen as a normalization of the cross-correlation. Intensities are normalized by subtraction of the mean intensity followed by a division by the standard deviation.

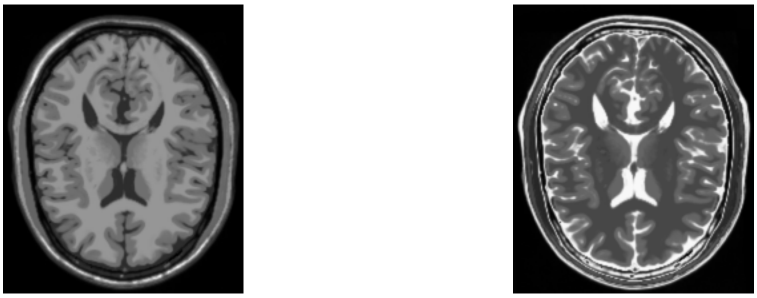


Fig. 2.1: Slices of simulated T1 (left panel) and T2 (right panel) MR images. The volumes are perfectly aligned, which is possible because they are simulated. T1 and T2 can be considered two different modalities.

Mutual information is a similarity metric based on information theory, which was first proposed for image registration independently by Viola and Wells [103], and Maes et al. [64]. The mutual information I of two images A and B can be defined in different ways. The definition

$$I(A, B) = H(B) - H(B|A) \tag{2.1}$$

bases on the Shannon entropy H. Entropy can be interpreted as a measure of uncertainty. Given events $e_1, e_2, \ldots e_n$ occurring with probabilities

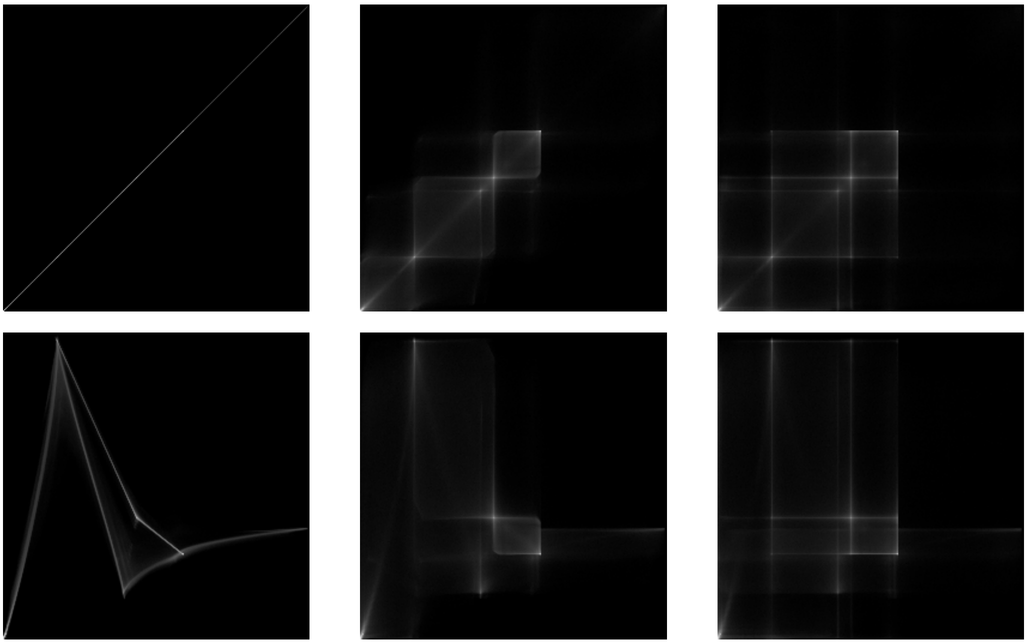


Fig. 2.2: The joint histograms for the intramodal case (two T1 images, top) and the intermodal case (T1 and T2 image, bottom) without deformation (first column), a translation by 2mm (second column) and a translation by 5mm (third column) demonstrate that misalignment results in a dispersion or blurring of the joint probability density function. The mapping of the intensities is linear in the intramodal case but not for the intermodal case. In the joint histogram images, the x-axis is the intensity in one image and the y-axis the intensity in the other image. A bright point stands for a high probability of the respective intensity pair, while a dark point implies a low probability.

$p_1, p_2, \ldots p_n$, the entropy is calculated as

$$H = \sum_{i=1}^{n} p_i \log \frac{1}{p_i}. \tag{2.2}$$

If all the p_i are equal, the entropy or uncertainty is highest. In our case, the events are the image intensities, and the interpretation of (2.1) is that the mutual information is the uncertainty about the image intensity in image B minus the uncertainty about the image intensity in image B if the intensity in image A is known. If the latter uncertainty is low, the mutual information is high and the images have a high similarity.

Another definition, based on the Kullback-Leibler distance for two distributions, is

$$I(A, B) = \sum_{i_a, i_b} p_{ab}(i_a, i_b) \log \frac{p_{ab}(i_a, i_b)}{p_a(i_a) p_b(i_b)}, \tag{2.3}$$

where p_a is the distribution of gray values i_a in image A and p_b the distribution of gray values i_b in image B. In this case, the distance between the joint distribution of the gray values p_{ab} and the joint distribution in case of independence ($p_a \cdot p_b$) is calculated. The joint distribution is obtained through normalization of the joint histogram. Figure 2.2 shows the joint histogram of aligned and misaligned BrainWeb [22] MR images (Figure 2.1). Under misalignment, the joint histogram becomes more blurred. Intuitively, mutual information is maximized if the joint histogram is “sharp”. Often, the registration problem is expressed as the minimization of the negative mutual information instead of the maximization of the mutual information.

While SSD and cross-correlation are popular metrics for intramodal registration, mutual information allows multimodal registration. This means that the similarity metric is an important design choice, because it can limit the applicability of the registration algorithm. Another important choice is the transformation model. A few of the most important ones are discussed in the following sections.

2.2 Physics-Based Approaches

The earliest reported nonrigid registration techniques base on physical models. An algorithm presented in 1989 by Bajcsy and Kovačič treats the moving image as a linear elastic solid [10]. It is deformed using forces derived from a similarity metric or landmark correspondences. These external forces are opposed by internal forces trying to prevent deformation of the moving image. The internal forces are derived from the elastic model. For the solution of the registration problem, internal and external forces reach an equilibrium. Because the internal forces increase proportionally with the deformation, this model makes it difficult to recover large deformations. Christensen et al. proposed a viscous fluid model [19], which allows large deformations but may also result in misregistrations due to the higher flexibility. More sophisticated models subdivide the image into cells and assign them local physical descriptions depending on the properties of the modeled structures [30, 38]. While soft tissue will have elastic properties, bone can be modeled as a rigid structure.

These models typically yield partial differential equations (PDEs). Either the finite element method (FEM) or the finite difference method is usually applied to solve them. For the finite element approach, a mesh (typically tetrahedral in 3D) is used to subdivide the continuous domain into a set of subdomains and approximate the solution. The deformation is defined at the nodes of the mesh and interpolated between the nodes. The discretization allows to replace the partial differentials with difference quotients (e.g. $f'(x) \approx (f(x+h) - f(x))/h)$. This yields an equation system which needs to be solved. Due to the size of the equation system, direct computation of its solution may not be feasible and iterative techniques may be required. An advantage of the finite element method is that the precision of the model can be varied, typically by selecting a finer mesh in "critical" regions and a coarser mesh otherwise.

For the finite difference method, a regular grid is placed over the continuous domain and the partial differentials are replaced with difference quotients (like for the finite element method). Starting from an initial estimate, the solution is found iteratively over discrete time-steps. The problem can either be solved using the explicit method or the implicit method. With the explicit method, the value at a certain grid point and time t_{i+1} is calculated directly from the values of the neighboring grid points at time t_i. With the implicit method, a system of linear equations of the values at the grid points at times t_i and t_{i+1} needs to be solved on each time step. The implicit method is usually more expensive but for the explicit method, step size and grid spacing need to fulfill certain constraints to converge.

Speed and robustness of these methods can typically be increased by means of a multiresolution strategy. For this purpose, the registration is first carried out with downsampled images (and coarser transformation models). The resulting transformation is upsampled to correspond to the resolution of the original images and used to initialize the transformation for the registration at this resolution. Multiple hierarchical levels are possible.

If the model is a good representation of the physical phenomenon that caused the image deformation, it may be the ideal approach to solve the nonrigid registration problem. However, in practice it may be very difficult to specify an accurate model. Furthermore, it yields a very problem-specific solution. Another issue is the computational power required for the registration. In 1996, Christensen et al. [18] reported registration to take up to more than a week on a single-processor system for a 128 × 128 × 100 voxel image. Although processor performance has increased considerably since 1996, so have image resolutions and therefore the complexity of registration problems. For these reasons, other image registration methods were proposed, of which some are discussed in the following sections.

2.3 Demons Algorithm

Thirion presented a method, which he illustrated by an analogy with Maxwell's demons [98, 99]. He describes the registration process as the diffusion of the moving image through the contours of the objects in the fixed image. The final transformation is found through iterative optimization, starting with an identity transformation. Maybe the most common variant of the demons algorithm (Demons 1 in [99]) selects all pixels of the fixed image to be demons. It uses a free-form transformation, meaning that the displacement is stored individually for each voxel of the fixed image. The corresponding moving image coordinate is the coordinate of the fixed image voxel plus the displacement vector. The displacement field is updated during each iteration based on the diffusion process. To get a smooth displacement field, it is regularized by applying a Gaussian filter with a given width σ after each iteration. A multiresolution strategy is adopted in order to improve speed and robustness of the algorithm.

The demons algorithm is reported to be very fast compared to the viscous fluid approaches described in the preceding section (20 minutes for 256 × 256 × 128 voxel images on a workstation PC). Bro-Nielsen and Gramkow [14] observe similarities to the viscous fluid registration, but with the Gaussian filter being a heuristic-based approximation of the physical model. They suspect problems in terms of topology and stability of the fluid model.

The demons algorithm assumes corresponding points in the two images to have the same intensity, which makes it unsuitable for multimodal registration. However, the underlying transformation model, which allows each point to move freely and imposes continuity by a smoothing filter, can also be combined with other similarity metrics. Guimond et al. propose an algorithm inspired by the demons algorithm, which optimizes mutual information [35].

2.4 B-Spline Model

In contrast to the fully local transformation model of the demons algorithm, B-spline transformations are semi-local. They are defined by a regular grid of control points, which are allowed to move freely. The dense transformation field (between the control points) is obtained by B-spline interpolation. The degrees of freedom of the transformation model (number of knots of the grid times the number of dimensions) depends on the grid spacing and the image size. The choice of the grid spacing is mainly an empirical one. B-spline transformations are smooth in general, but a finer grid allows more local transformations. The parameters of the model are the B-spline coefficients, located at the control points.

B-splines have a limited support: modifying one parameter only changes the transformation field in the vicinity of the respective control point. This property enables the efficient computation of the transformation function at arbitrary points, independent of the total number of parameters of the model. There are B-splines of different degrees. A higher degree yields a smoother function but also a more complex model because the B-spline functions have a larger support. Usually cubic B-splines are used, which evaluate to nonzero values in the range $]-2;2[$. This means that the transformation at a specific point is defined by a neighborhood of 4^N control points, where N is the number of dimensions.

This transformation model has been used in conjunction with different similarity metrics, for example SSD [57, 95] or mutual information [67, 86]. Usually, the parameters are initialized to zero and iterative optimization algorithms are applied to find a set of parameters yielding the highest similarity. Similar to previously discussed algorithms, a multiresolution strategy may provide better robustness and speed. The coarse-to-fine strategy can be applied both to image resolution and control-point spacing.

2.5 Piecewise Rigid Registration

Other approaches consider the globally nonrigid registration problem as a number of semi-local rigid registration problems. Little et al. [61] proposed a transformation model, where a number of selected structures in the body are deformed rigidly, each one with an independent transformation. A smooth transformation field based on landmarks is used to transform the points outside of the rigid structures. An approach by Maintz et al. [66] divides the moving image into small "windows" after a global rigid registration. Small translations for each window are searched in order to compensate local deformations. This scheme was extended by Likar and Pernuš [60]. In their approach called volume-subdivision strategy, the images are progressively subdivided into smaller images, which are registered rigidly. For example, after rigid registration of the entire 2D images, they are split into four rectangular subimages of half the width and height of the original images and so forth. A continuous, smooth transformation function is obtained by a thin-plate spline interpolation using the centers of corresponding subimages as landmarks.

This scheme allows to use local image statistics to compute the mutual information (other approaches usually compute the global metric). This may be beneficial if there is no global relationship between the image intensities, for example in the presence of a bias field yielding smoothly varying intensity values in regions of MR images which should be homogeneous, shadowing (a structure covering the ones behind it) in ultrasound or shading effects due to varying slice thickness in microscopy. However, the subimages possibly need to be relatively large in order to provide sufficient samples for the estimation of the joint probability density function [8]. This can prohibit deformations on a very small scale.

3

Accelerating Nonrigid Image Registration

3.1 Overview

In this section, implementations of nonrigid registration algorithms designed to reach low runtimes are presented. Acceleration usually was achieved by exploiting parallelism. Traditionally, high-performance computing systems, such as clusters and shared-memory multiprocessors, have been exploitet for this purpose. These approaches are addressed in a first section of this chapter. During the work on this dissertation, research started to increasingly focus on systems providing better performance-per-cost ratios, especially GPUs (graphics processor units). This is due to certain trends in processor development, which are addressed in a next section. The remaining sections give an overview of a number of such platforms and registration algorithms running on them.

Speedup and efficiency are two important means for the discussion of the performance of parallel algorithms. The speedup when using p processors,

$$S_p = \frac{T_1}{T_p}, \tag{3.1}$$

depends on the runtime when using one processor (T_1) and the runtime when using p processors (T_p). Ideally S_p is equal to p. The efficiency

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \tag{3.2}$$

ideally is equal to 1. Typically, algorithms do not scale linearly and the efficiency decreases for higher p.

3.2 Multiprocessors

Multiprocessing, where two or more central processing units (CPUs) are used within one computer system, is a common parallel computing approach. Multiprocessor systems fall into two categories: shared-memory and message-passing architectures. In this section, a brief overview is given. The interested reader is referred to [26].

3.2.1 Shared-Memory Architectures

Although there is a wide variety of shared-memory architectures, they have one important feature in common: All the processors can access the entire memory as a global address space. The processors in the system can operate independently but share data structures in memory. This communication model provides a programmer-friendly environment. No explicit data exchange is necessary. Since all communication and local computations generate memory accesses, the key architectural high-level design decision is the organization of the memory hierarchy.

The bus-based shared memory approach is an example of an UMA (uniform memory access) architecture. UMA means that all of main memory is equally far away from all processors. The processor interconnect is a shared bus between the processors' private caches and the shared main memory. This architecture has been widely used for relatively small numbers of processors (up to 20 or 30). Further scaling is primarily limited by the bandwidth of the interconnect and the memory system.

An approach intended to be more scalable is the NUMA (non-uniform memory access) architecture. An interconnect is placed between the processing nodes. Each processing node has its own portion of the global memory. Access to memory "owned" by other processors go through the interconnect and therefore is slower than access to directly connected memory.

While NUMA architectures overcome some of the scalability issues of UMA architectures, a problem inherent to all shared-memory architectures is the cache coherency. Furthermore, synchronization is required in the case of concurrent data accesses by different threads.

Coherency

		contents for location X		
Time	Event	Cache P1	Cache P2	Memory
0				1
1	P1 reads X	1		1
2	P2 reads X	1	1	1
3	P1 stores 0 to X	0	1	0

Fig. 3.1: The cache coherence problem for a memory location X shared by two processors P1 and P2. Initially, neither cache contains the value of X. After the value of X has been written by P1, its cache and the memory both contain the updated value (assuming a write-through cache). However, if P2 reads X again, it will receive the outdated value from its cache.

Processors replicate shared data in their caches for faster access. Because each processor sees global memory through its individual cache, two different processors can have two different values for the same memory locations (see figure 3.1). There are hardware solutions that allow maintaining coherent caches, called cache coherence protocols [40]. A write-invalidate protocol invalidates copies in other caches on a write by a processor. The alternative to invalidating is updating all the cached copies. This type of protocol is called a write-update protocol. With increasing number of processors, the traffic associated to the cache coherency protocol grows, which limits the scalability of such systems.

Synchronization

Although the shared-memory communication model is relatively convenient for the programmer, it requires synchronization. On multiprocessors, multiple threads[1] are running in parallel on the different processors, possibly with different speeds. When different threads work on shared data, synchronization operations may be required to enforce the order of accesses by different threads. There are two basic types of synchronization, mutual exclusion and events.

t	Op P1	Op P2	X
0	load X		0
1	X = X+1		0
2	store X		1
3		load X	1
4		X = X+1	1
5		store X	2

t	Op P1	Op P2	X
0	load X		0
1	X = X+1		0
2		load X	0
3	store X		1
4		X = X+1	1
5		store X	1

Fig. 3.2: If two threads want to increment a shared variable, the result depends on the order, with which the load and store operations are executed. While the programmer may expect the result in the left panel, incorrect synchronization may yield the result shown in the right panel. Synchronization of the threads is necessary to ensure correct behavior.

Mutual exclusion ensures that only one thread modifies certain data at a time. If, for example, two threads share a counter X, which each thread wants to increment, the result can depend on the order, with which the operations are executed (Figure 3.2). The code which modifies the counter is called the critical section. Only one thread is allowed to be in the critical section of the program at a time. Mutual exclusion operations tend to serialize the execution of processes and therefore limit the possible speedup of parallelization.

Parallel programs are usually a series of (sequential and parallel) stages. Before the next stage can be entered, all threads have to finish the previous stage. Events are the synchronization mechanism used to inform other threads that a certain point in the program has been reached. When parallelizing an algorithm, one has to distribute the workload evenly among the threads, which is not always easy. Load imbalances result in threads waiting for others to finish before the next program stage can be entered.

Synchronization operations typically base on an interplay of hardware

---

[1] A thread is a sequence of instruction. Multithreading means that multiple threads are executed in parallel.

and software. Higher level synchronization mechanisms can be implemented in software based on *atomic read-modify-write* operations. Typically, a programmer wants to use such high level mechanisms without caring about their internal implementation. There are many software libraries and APIs (application programming interfaces) providing synchronization and a selection will be presented in section 3.2.3.

3.2.2 Message-Passing Architectures

An alternative to having one shared-memory space is organizing the memory as multiple private address spaces. A processor cannot directly access the memory of a remote processor. The processor-memory modules can even be separate computers, connected by a network. Such *clusters* built from off-the-shelf computers can offer very cost-effective parallel programming environments. Communication occurs by explicitly sending and receiving messages across the network. On clusters, processors are coupled more loosely than on shared-memory architectures. They are usually connected using the I/O bus. This bus usually has a lower bandwidth and a higher latency than the memory bus, which is the typical interconnect for shared-memory architectures.

Hybrid architectures are also possible, for example by connecting multiple shared-memory systems over a network. Such setups become more and more popular as nowadays even normal workstation PCs are typically equipped with multicore processors.

3.2.3 Programming Models

Grama et al. [32] wrote a comprehensive introduction to parallel programming. Among the available parallel programming APIs, POSIX (Portable Operating System Interface) threads and OpenMP (open multiprocessing) for shared-memory architectures and MPI (message passing interface) for message-passing architectures will be presented in this section.

POSIX Threads

POSIX threads (or Pthreads) is a standard, which defines an API for creating, modifying and synchronizing threads on a shared-memory architecture. This API is common on Unix-like operating systems, like Linux, but also available for Microsoft Windows. The Pthreads API defines a set of types and functions for the C programming language.

Listing 3.1: Pthreads Example

```
#include <stdio.h>
#include <pthread.h>

void *func(void *arg) {
        printf("Hello␣world!\n");
}

int main() {
        pthread_t thread;
        pthread_create(&thread, NULL, func, NULL);
        pthread_join(thread, NULL);
}
```

Listing 3.1 shows a Pthreads example program. The function `pthread_create` spawns an independent thread, which executes the function `func` and then terminates. The main program waits for the thread to finish (`pthread_join`) and exits.

Furthermore, functions to create and manipulate mutexes (mutual exclusions) are defined in order to allow mutual exclusion within critical sections of the program. Mutexes work like locks. Only one thread can lock (or own) a mutex at a time.

Condition variables allow event synchronization of threads. If one thread `T1` needs to wait for a result from another thread `T2`, `T1` can wait on a condition variable defined for this purpose. The thread is “sleeping” while it is waiting and thus not consuming resources. When the result is ready, `T2` can signal on the condition variable, which causes `T1` to wake up and continue working.

OpenMP

Use of Pthreads is mostly restricted to system programmers because it is considered low-level. OpenMP is a higher level API for shared-memory architectures. It frees the application programmer of the low-level thread administration, such as spawning and joining of threads and explicit setup of mutexes and condition variables. It is based on `#pragma` directives. The program consists of sequential and parallel stages. As soon as the beginning of a parallel stage is reached, indicated by a `#pragma omp parallel` line in the code, a number of threads, which can be specified, is spawned. The sequential thread becomes the master thread and the new threads its slaves. The master thread joins the slaves at the end of the parallel stage and continues with the next sequential stage.

Listing 3.2: OpenMP Example

```
#pragma omp parallel {
  #pragma omp for schedule(static)
  for (i = 0; i < N; i++) {
    ... //do work
  }
}
```

In the example in listing 3.2, a loop of N iterations is parallelized. The `schedule(static)` statement indicates that each of the threads should process an equal share of the N iterations, which makes especially sense if all the iterations take about the same time. If this is not the case, dynamic scheduling is preferable to balance the load efficiently among the threads. However, the programmer still has to declare which variables are private (each thread has its own copy) and which ones are shared. Furthermore, in the previous example he has to make sure that each iteration of the `for` loop is independent of the others.

MPI

MPI (message passing interface) is a communication API for architectures without a shared address space. The threads exchange data and synchronize by sending messages over the interconnect. The functions `MPI_Send` allows a thread to send data to another thread. The destination thread has to call the function `MPI_Recv` to receive the data.

MPI also defines more complex communication patterns, such as `MPI_Barrier`, `MPI_Broadcast` and `MPI_Reduce`. The `MPI_Barrier` function allows to synchronize a group of threads. It causes all the threads in the group to wait until each one has reached the call of the barrier function. The `MPI_Broadcast` function allows one thread to send data to a group of threads.

The `MPI_Reduce` function is a combination of communication and computation. It allows to efficiently combine data from all the threads in the group. The threads collectively exchange data and perform a operation on it, for example calculating the sum of a variable, where each processor has its own private copy.

3.2.4 Early Nonrigid Registration Work

Multiprocessors were used for for nonrigid registration as early as 1996 by Christensen et al. [18]. Their massively parallel implementation is used to map a brain atlas to patient 3D MRI data. It uses a viscous fluid model in conjunction with the SSD similarity metric. The resulting partial differential equation is solved iteratively using finite differences. For $128 \times 128 \times 100$ voxel images, the authors measured a sequential runtime of 1.4 days to more than 7 days, depending on the processor (MIPS R8000 and MIPS R4400 respectively) . On the processor requiring 1.4 days, 6.4 hours of the runtime are attributed to computations related to the SSD similarity metric. The larger part, 1.1 days, are attributed to the regularization term based on the viscous fluid model.

To reduce registration time, the algorithm was implemented on MasPar, a SIMD (single instruction multiple data) system with 16384 processor elements arranged on a quadratic grid. All the processor elements execute the same operation, but on different data. Only neighboring processor elements can communicate efficiently. Computation of the regularization term can be mapped very efficiently on this architecture. This stage of the algorithm is accelerated by 57× compared to the sequential implementation and its runtime is reduced to 28 minutes. The computation of the similarity metric requires the transformation of the atlas image. For large transformations, this can result in much more complex communication patterns than for the regularization term and its computation on the parallel system is not very efficient. Therefore, this stage is only accelerated by a factor 5× and still takes 1.3 hours on the parallel system.

3.2.5 Biomechanical Models

More recently, image guided neurosurgery has triggered numerous research projects in the field of parallel image registration. Preoperative images are fused with intraoperative images to augment the information provided to the surgeon. The preoperative images may have a higher spatial resolution or a better signal-to-noise ratio. Furthermore, they may be of a different modality and therefore provide different information. In such an environment, fast registration is crucial because the results need to be available within a reasonable time.

Warfield et al. [104] establish correspondence based on an explicit representation of the surface of the brain and the ventricles. For this purpose, a manual segmentation of these structures in both images is required. Then a tetrahedral mesh covering these structures is generated. The surfaces of the brain and the ventricles from the preoperative image are deformed iteratively until they match the surfaces from the current image. The position of the surface nodes of the tetrahedral mesh are set according to this deformation. The deformation within the volume is found by deforming the mesh under constraints of an elastic body model. Solving the resulting equation system is identified as the most time consuming task. It is parallelized using the *Portable, Extensible Toolkit for Scientific Computation* package, which itself bases on different other packages, such as MPI for the message passing communication. The solution time for a mesh with 43584 nodes (or 214035 tetrahedral elements) is reduced from 57 to 15 seconds on a shared memory architecture with 12 UltraSPARC-III processors. However, the total runtime of the method is bound by the segmentation step, which can take several minutes. Workload balancing is a major problem for the efficiency of the parallel implementation. The authors expect that a mesh with a more regular connectivity pattern would allow better scaling.

Sermesant et al. address the same problem using a similar mesh structure with a biomechanical model of the brain [88]. Instead of surface extraction, they propose to use 3D block matching around the vertices of the mesh. Block matching tries to find for a block of the fixed image (here the neighborhood of a vertex of the mesh) the most similar block in the moving image [76]. The correlation coefficient is used as the similarity metric. Establishing correspondence for all the vertices of the mesh allows to estimate more complex deformations inside the model, which are not limited to interpolation of the surface deformation. For parallelization, each vertex is assigned to a processor, such that communication cost is minimized. A scalability analysis using the BrainWeb images [22] ($181 \times 217 \times 181$ voxels) and a mesh with around 2000 vertices is carried out on a cluster consisting of 14 processors interconnected by Gigabit Ethernet. The speedup of almost 6× when using eight processors improves only to around 7× when using all the 14 processors. The 14 processor setup yields a registration time below three minutes. The authors attribute the rapid decrease in efficiency to the relatively coarse mesh used in this study.

Chrisochoides et al. implement a similar algorithm and identify the block matching phase as the main bottleneck [17]. This phase is parallelized on a heterogeneous system with 240 processors, which results from connecting multiple smaller clusters. On each cluster, there is a master thread. The master threads use TCP (Transmission Control Protocol) for communication. Intra-cluster communication is provided by MPI. Load balancing is a main issue because it limits the scalability of the block matching phase. Problems are the different performance of the processors and the fact that certain machines may be used by other users. Therefore, a dynamic load balancing method is implemented, which allowed to reduce the runtime of the block-matching stage to around 30 seconds. But including the sequential pre- and postprocessing steps, such as the finite element solver, the overall runtime is still in the range of 160 to 200 seconds.

3.2.6 Demons Algorithm

Stefanescu et al. implemented the demons algorithm on a Gigabit-Ethernet cluster of 15 PCs [92]. With respect to the runtime, one of the main parts of this algorithm is the Gaussian filtering of the dense transformation field, which has to be executed during each iteration of the algorithm. The authors approximate this step by a separable, recursive filter. For parallelization, the image is distributed slice-wise to the processors. At the borders of the slices, communication between processors is required during recursive filtering. Two implementations are proposed. For the first one, a small number of large data blocks have to be exchanged. The size of the blocks increases with the standard deviation of the Gaussian. This implementation is efficient for small standard deviations and high-latency networks. For the second implementation, many small blocks of data need to be exchanged. The communication scheme is independent of the standard deviation of the filter, but has drawbacks on high-latency networks. For a pair of $256 \times 256 \times 120$ MR images, a reduction of the registration time by a factor of $11\times$ from 40 minutes on one processor to 3.5 minutes is measured.

3.2.7 B-spline Algorithms

A parallel implementation of a B-splines based registration algorithm is presented by Rohlfing et al. [85]. It runs on shared-memory platforms and bases on Pthreads. Mutual information is the similarity metric. For the performance analysis, the algorithm was applied to three different registration problems: intraoperative brain deformation analysis, motion correction in contrast-enhanced MR mammography and intersubject brain atlas generation. For $512 \times 512 \times 60$ voxel images, a sequential runtime of around 3 hours is reported. For 16 processors, the runtime is reduced to 13.5 minutes (13.8× speedup). Using 64 processors of an SGI Origin 3800, 4.7 minutes were measured (39× speedup). The sequential part was estimated to be around two minutes, yielding a maximum possible speedup of 89×.

Not much later, Ino et al. presented an MPI based implementation of a very similar algorithm [47], running on a 2 Gbit/s Myrinet cluster of 64 off-the-shelf dual-processor (Pentium III) workstations. Using all 128 processors, the registration time for a $512 \times 512 \times 159$ voxel image is reduced from 12 hours to eight minutes (90× speedup).

3.2.8 Conclusions

Multiprocessor systems achieve runtimes in the order of a few minutes for different nonrigid registration algorithms. Typically, the performance of the parallel algorithm is assessed by comparison to the runtime of the same algorithm running on a single processor. It is difficult to compare the different implementations. First, it is virtually impossible to compare the quality of the results. Iterative methods may have a parameter defining an upper limit to the number of iterations. Because the runtime is usually proportional to the number of iterations, it can be limited by this parameter, at the price of a possible loss of accuracy. Some algorithms use one or more criteria to detect convergence of the iterative optimization process. However, how convergence is defined again has an influence on accuracy and runtime.

For intensity based approaches, the runtime also depends heavily on the image size. Because their images are more than 2.5× larger, it is not so surprising that Ino et al. report a higher registration time than Rohlfing et al. although they use a larger system. We can also observe that Ino et al. achieve an efficiency of 0.75 for 128 processors, while Rohlfing et al. report efficiencies of 0.86 for 16 processors and 0.61 for 64 processors. Because two different sequential implementations are used as the baseline, the algorithm that has the higher efficiency is not necessarily faster in absolute terms.

However, the discussed multiprocessor solutions generally base on systems which are relatively expensive in acquisition and maintenance. In the next section, we will discuss trends in processor architecture. There are a number of emerging architectures providing a considerable computational performance at a much lower price.

3.3 Trends in Processor Development

Historically, CMOS (complementary metal-oxide-semiconductor) transistor density has doubled every 18 months. Thus, each new generation also allowed to build more powerful processors, on the one hand by ramping up the clock frequency and on the other hand by increasing the complexity of the processors. For example, more and more inherent instruction-level parallelism was exploited.

3.3.1 Instruction-Level Parallelism

Since about 1985, all processors use pipelining to overlap the execution of instructions, a key technique to make fast processors. For this purpose, execution of an instruction is subdivided into multiple stages. A simple RISC (Reduced Instruction Set Computing) architecture could use the following five stages, each one executed in one processor cycle [40].

1. *Instruction fetch cycle*: Fetch the current instruction from memory and increment the program counter.
2. *Instruction decode/register fetch cycle*: Decode the instruction and read the registers corresponding to the register sources from the register file.
3. *Execution cycle*: The ALU (arithmetic logic unit) operates on the operands prepared in the preceding cycle.
4. *Memory access cycle*: Load or store data from/to memory.
5. *Write-back cycle*: The result (from an ALU operation or a memory load) is written to the register file.

Executing the entire instruction still takes around the same time, except for some overhead due to unbalanced pipeline stages and additional latches. However, the processor can issue a new instruction every cycle. When the first instruction is handed over to the second pipeline stage, a new instruction can enter the first stage. Instruction-level parallelism increases with pipeline depth. Deeper pipelines are only beneficial as long as the gain in number of instructions one can issue in an amount of time exceeds the time lost due to penalties associated to a pipelined design [91].

In addition to the penalties already mentioned earlier, there are situations, called hazards, that prevent the next instruction from being issued. *Structural hazards* are resource conflicts that can occur if the hardware does not support all possible combinations of overlapping instructions. *Data hazards* occur if an instruction depends on a result by a preceding instruction, which has not yet finished. A load instruction, for example, can take hundreds of cycles to complete if the data to be loaded is not cached. *Control hazards* are related to pipelining of branches and other instructions that change the program flow. It is not clear which instruction will be the next one to be placed into the pipeline until the branch instruction finishes.

During the evolution of processors, pipelines became increasingly deep and additional hardware was added in order to control penalties. Branch prediction and dynamic reordering of independent instructions are examples. Pipelining combined with these techniques yields an ideal CPI (cycles per instruction) of one. The goal of *multi-issue processors* is to further reduce CPI below one. The basic multi-issue processors are *superscalar* processors and *VLIW* (very long instruction word) processors. Superscalar processors issue varying numbers of instructions to different pipelines. VLIW processors issue a fixed number of instructions formatted as one large instruction.

Because of the added complexity associated to increased instruction-level parallelism, doubling the transistors typically only meant an (integer) performance benefit of around 1.5× [83]. Furthermore, some of these techniques provide diminishing returns [44].

At some point it became clear that sustaining the historical performance growth rate was not possible based on aforementioned techniques only [13]. The limiting factor shifted from manufacturability to power consumption, which today has surpassed 100W for some processors. Considering the small size of processor chips, it becomes increasingly difficult to supply the required power and control the temperature of the chip. Power consumption could be reduced by reducing supply voltage, clock frequency or die size (and thus transistor count). All of these approaches reduce performance, therefore one has to trade off performance to reduce power consumption. In order to consider the relation between computational performance and power during the processor design process, new design goals for microprocessor architectures were proposed [43]. These goals changed the direction of microprocessor development.

3.3.2 Multicore Processors

If between two processor generations the number of transistors doubles, another way to use the additional transistors is to place multiple identical processor cores on one chip. Depending on the workload, this can almost double the performance of the processor and thus is a potentially more efficient approach than the traditional ones. However, the application programmer needs to explicitly parallelize the software in order to exploit the potential of multicore processors. In early 2009, quad-core processors already have been on the market for a while, and different six and eight core processors are on the horizon. A consequence of adopting multicore processor designs is that single-thread performance cannot be expected to increase much from one generation to the next, because the individual cores do not get much more powerful any more.

As a matter of fact, some designs try to reclaim power efficiency more aggressively. By returning again to simpler in-order cores, it is possible to pack even more of those onto one chip and further increase overall computational throughput. Intel presented a manycore architecture code-named Larrabee [87]. It targets visual computing and bases on in-order x86 processors with a simple pipeline and a wide vector-processing unit. Researchers are already investigating architectures and programming models for manycore processors with hundreds of cores [39].

While manycore architectures certainly present many challenges to the programmer, there are also opportunities. On-chip buses for the intercore communication provide lower latencies and higher bandwidths than the interconnects used in multiprocessor systems.

3.3.3 Asymmetric Processors

As a consequence of the simpler cores, many-core processors will have lower single-thread performance. This hampers performance of legacy applications that are not parallelized, but possibly also has an impact on performance of parallelized software. Parallel algorithms usually still have a more or less small sequential fraction. To allow a trade off between sequential peak performance and total throughput, asymmetric architectures consisting of multiple cores of different sizes are possible. Hill and Marty [42] show that for a potential 256 core processor, sacrificing some of the small cores and replacing them with a powerful superscalar core can boost the performance even for applications with relatively small sequential parts. However, they do not consider power consumption in their models. Woo and Lee [105] come to the conclusion that an asymmetric processor with many small, energy-efficient cores combined with a full-blown processor also delivers the best energy efficiency.

3.3.4 Accelerators

Special-purpose hardware units, customized for a certain task, are another way to increase power efficiency (and area efficiency). Commercially successful accelerator chips are available for domains like graphics, gaming, network processing (for example TCP offloading, encryption, deep packet inspection and XML (extensible markup language) processing), and video encoding [79]. While these accelerators traditionally were implemented as separate chips attached to the CPU, it becomes more common to couple them more tightly to the processor by placing them on the same die. Like this, the accelerator can be seen as a heterogeneous extension to the base platform. Depending on the nature of the accelerator, this approach may be similar to the asymmetric processors with cores of different sizes.

3.3.5 Memory

Semiconductor computer memory also benefits from advances in silicon technology. Its capacity grows rapidly, similar to the computational performance of multicore processors. However, commodity DRAM (dynamic random-access memory) performance, measured in bandwidth (amount of data transferred per time) or latency (time between a data request and availability of the data), grows at lower rates. Bandwidth improves by about 25% per year, latency only by 5% [77]. This difference in growth rate makes it necessary to constantly rethink assumptions behind processor and system design.

For example, latency tolerance will become more and more important. Latency tolerance means limiting the time the processing units stall while latency to memory grows. Another goal is the reduction of off-chip communication by only fetching and storing really global data and keeping local data in the caches. Therefore, larger caches can also reduce bandwidth requirements.

3.3.6 Programming

The trend toward multicore and many-core processors means a shift in complexity from hardware to software. While hardware used to extract instruction-level parallelism from sequential programs, application programmers are expected to explicitly parallelize their software using similar programming models as discussed in the preceding sections about high-performance computing on multiprocessors. Otherwise, they cannot benefit from the new architectures [94]. The programming effort may be reduced by implicit parallel programming, where a sequential program with additional hints about dependencies is provided to the compiler [46]. However, these techniques are not yet mature and are not expected to efficiently parallelize legacy code. The programmer is still expected to recognize if an equivalent, more parallel algorithm is available for a certain problem. He may also understand the trade-offs necessary to choose a more parallel but not exactly equivalent algorithm.

The platforms presented in the following sections are related to the discussed topics. The Cell Broadband Engine (Cell/B.E.) [48] chip is an asymmetric (or heterogeneous) multicore processor consisting of one general purpose processor (the PPE — Power Processor Element) and eight SPEs (Synergistic Processor Elements). The SPEs are simpler cores, which are for example not capable to run an operating system. Therefore, they can also be seen as on-chip accelerators. GPUs (Graphics Processing Unit) traditionally were very specific accelerators, mainly optimized for rendering a 3D scenery to a 2D image. Because of their increasing applicability for broader tasks, they can also be considered off-chip accelerators. Finally, FPGAs (Field Programmable Gate Arrays) are reconfigurable devices, suitable to implement a wide range of accelerator units. They can be coupled more or less tightly to the CPU.

3.4 Cell/B.E.

3.4.1 Overview

The Cell/B.E. processor was designed with the objective to maximize computational performance per Watt. It was intended to especially excel at applications in gaming and multimedia [48]. Sony, Toshiba and IBM developed it collaboratively to drive Sony's PlayStation3 gaming console. But from the beginning, other application areas were targeted as well. The design is general enough for many workloads requiring heavy number crunching. An open software development kit (SDK) based on Linux and the GNU compiler collection (GCC) was provided to the software development community in order to foster optimization of numerous applications for this platform. After the first incarnation of the Cell/B.E. processor, which was released in 2005 in 90nm technology, a revised variant in 65nm technology called *PowerXCell 8i* was released in 2008. The main difference was the improved double-precision floating point performance, which was a weakness of the original processor.

3.4.2 Processor Architecture

The Cell/B.E. processor is a heterogeneous multicore processor, combining one PPE (Power Processor Element) and eight SPEs (Synergistic Processor Elements) on one chip (Figure 3.3). The on-chip element interconnect bus (EIB) connects the processor elements. The on-chip memory controller and I/O bus interface controller are also attached to the EIB. The memory bandwidth is 25.6 GByte/s. Peak I/O bandwidths are 35 GByte/s outbound and 25 GByte/s inbound. The I/O interface is configurable as two separate logical interfaces, for example one providing 30 GByte/s outbound and 20 GByte/s inbound and another one 5 GByte/s outbound and inbound.

3.4.3 PPE

The PPE is a processor based on the IBM PowerPC Architecture. Programs compiling for this architecture also run directly on the Cell/B.E. processor. However, to exploit the full potential of the chip, parts of the program need to be moved to the SPEs. In typical programs, the PPE is responsible for overall control and management of the SPEs as well as execution of tasks, which are not performance critical or difficult to parallelize.

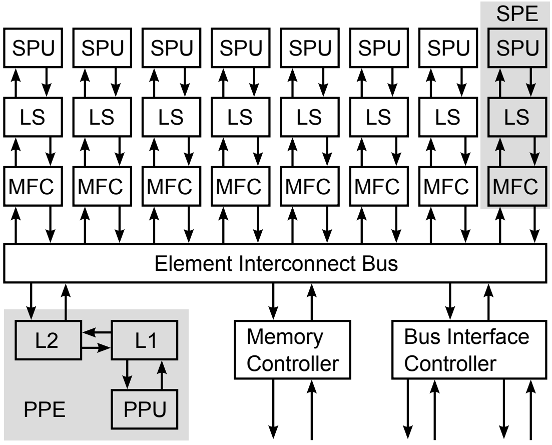


Fig. 3.3: Cell/B.E. processor. The Power processor element (PPE) is a general purpose processor. It has 32 kByte first-level (L1) instruction and data caches and a 512 kByte second-level (L2) cache. Each of the eight synergistic processor elements (SPEs) contains a synergistic processor unit (SPU), a 256 kByte local store (LS) and a memory flow controller (MFC).

Compared to other superscalar processors of the same generation, some features were omitted in the PPE. It is a dual-issue design compared to available four-issue designs. Furthermore, it is not an out-of-order processor. However, there is a separate, decoupled pipeline for floating-point and vector instructions. This allows reordering of instructions in one pipeline with respect to instructions in the other pipeline, if there are no dependencies. To optimize the use of the issue slots, two threads can be processed in an interleaved fashion.

The PPE is equipped with vector multimedia extensions (VMX) for 128-bit SIMD with variable element width. (e.g. 4 × 32-bit, 8 × 16-bit, …). For both threads, there is a 32-entry by 128-bit vector register file.

A conventional cache hierarchy with 32 kByte L1 (level 1) instruction and data caches and a 512 kByte L2 (level 2) cache is used.

3.4.4 SPE

The other type of processor element, the SPE, comprises an SPU (synergistic processor unit), a 256 kByte local store (LS) and a memory flow controller (MFC). It was designed with the goal to optimize computational performance per Watt (and area). Although more complex cores could provide more performance, this would mean that fewer cores could be placed on one chip due to the larger size.

The SPU is organized entirely around a 128-bit data path [34]. Vector and scalar data are handled in this data path. The overhead of having separate pipelines and issue slots for scalar and vector instructions was removed. As a consequence, the SPU is especially strong at running vectorized code. Instructions on scalar data occupy an entire issue slot, which could otherwise be used for a "more productive" vector operation. Data is loaded and stored in 128-bit granularity and at 128-bit aligned addresses. A penalty is associated to working on unaligned vector data. A large 128-entry 128-bit unified register file is shared for all types of data (scalar, vector, float, integer, ...). The size of the register file should facilitate efficient instruction scheduling by the compiler and enable loop unrolling.

There is no hardware support for branch prediction, but the programmer (or compiler) can provide branch hints. Upon notification of an upcoming branch, the hardware prefetches at least 17 instructions starting at the branch target address.

There is a single program running at a time on the SPU (no multithreading support). It only has direct access to the LS, which therefore has to accommodate program instructions and data. The LS is not part of the cache-coherence domain. Data are transferred between LS and system memory by explicit DMA (direct memory access) commands, issued by the program running on the SPU or the PPE and executed by the MFC. Typically, data are loaded by DMA, processed, and returned again to main memory by DMA. Addresses can be exchanged between PPE and SPEs because the same address translation mechanisms are used. The LS is mapped into global memory space in order to allow local-store-to-local-store transfers.

The MFC can work in parallel to the SPU. Once the transfer is set up, the SPU can continue working and polling (blocking or nonblocking) for transfer completion. Each SPE supports up to 16 outstanding DMA transfers, helping to avoid latency-induced bandwidth limitations. Supporting only a limited number of outstanding transfers means that preceding transfers need to finish before new ones can be issued. If the number of transfers in flight is low and each transfer carries little data, only a fraction of the actual bandwidth can be used.

3.4.5 On-Chip Communication

The PPE and the SPEs are connected by the element interconnect bus. Furthermore, the memory controller and I/O bus interface controller are also attached to the EIB, which enables communication among all these components [52]. It consists of four 16-byte wide data rings, two running clockwise and two counterclockwise. The EIB arbiter processes transfer requests and assigns a ring, such that the transfer does not need to go more than halfway around the ring.

One form of communication are the DMA transfers mentioned in the preceding section. The MFC supports naturally aligned[2] transfers of 1, 2, 4, or 8 bytes, or 16-byte aligned transfers with a length from 16 Bytes to 16 kBytes. Both source and destination address need to be aligned.

Other means of communication are signal notification and mailboxes. Each SPE has two incoming 32-bit signal notification channels. The other SPEs and the PPE can write to these channels using memory-mapped addresses. This is a very simple and fast way of communication. These registers can be configured in *OR* mode, meaning that the result of a write operation is a bit-wise OR of the original content of the register and the written value. This allows collective communication involving multiple processing elements. Messages from different SPUs or the PPE can be received in the signal notification register, if the affected bits do not overlap. Used this way, the order with which signals are received does not matter.

Each SPE also has two four-entry outbound mailboxes and one four-entry incoming mailbox. They are suitable for point-to-point communication, as in master-slave or producer-consumer models and are slower than signal notification. If an incoming mailbox is full, the SPE or PPE that wants to write to it has to wait until the owning SPE frees a slot by reading an entry. Similarly, the owning SPE cannot write to a full outgoing mailbox until a consumer frees a slot by reading a message.

3.4.6 System Architecture

While the PlayStation3 gaming console is the first commercial application based on the Cell/B.E. processor, there are alternative devices and systems. Mercury Systems offers an accelerator board, which can be attached to a

---

[2] A data block is called naturally aligned if its address is aligned to its size, for example a 8 byte data structure located at an address being a multiple of 8.

standard PC by a PCIe (peripheral component interconnect express) link. The board is equipped with four GByte of memory. The largest Cell/B.E. based system, the Roadrunner supercomputer, is also leading the list of the TOP500 supercomputer sites[3] [71]. It consists of 3060 nodes, each featuring four PowerXCell 8i processors and two AMD Opteron dual-core processors [11]. These nodes, the triblades, are interconnected using InfiniBand[4]. A triblade consists of three blades, one for the Opteron processors and two equipped with two PowerXCell 8i processors each. Blades are computer systems equipped with all the necessary components but with a physical design minimizing the required space. Multiple blades can be mounted in one enclosure (or chassis), which can provide services like power supply, cooling, storage, and networking.

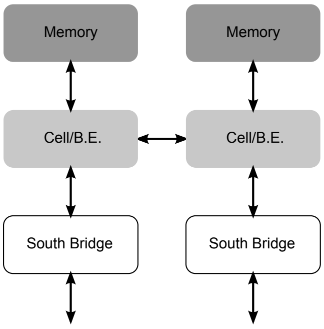


Fig. 3.4: A Cell/B.E. blade is equipped with two processors, connected directly by a coherent interface. Memory is directly connected to each processor. I/O functionality is provided by a separate chip, called South Bridge or I/O controller hub.

The Cell/B.E. blades used for Roadrunner are commodity systems, which can also be used stand-alone. They are another example of a Cell/B.E. based system [73]. The two Cell/B.E. processors are connected gluelessly (without any additional switch chips) by a coherent interface to form a shared-memory multiprocessor (Figure 3.4). For this purpose, one of the I/O interfaces of the Bus Interface Controller is reconfigured to serve as processor interconnect. The bandwidth is 20 GByte/s in each direction. Although the memory is shared by the two processors, each one has faster

---

[3] June 2009 revision — the latest at the time of the writing

[4] InfiniBand is a communication link mainly used in high-performance computing.

access to its own memory, meaning that the Cell/B.E. blades are shared-memory NUMA systems.

System software bases on a 64bit PowerPC SMP-enabled (symmetric multiprocessing) Linux kernel with additional patches to support the SPEs. The operating system schedules the applications on the two processor and decides on which physical SPE to run an SPE kernel. However, a PPE application can use the SPEs on the other processor transparently and spawn up to 16 SPE threads.

3.4.7 Programming Models

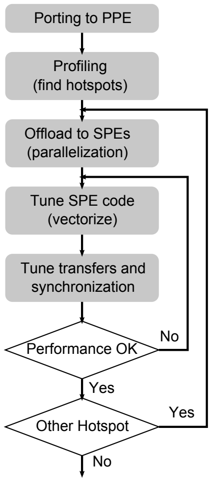


Fig. 3.5: Porting an algorithm to the Cell/B.E. architecture typically is an iterative process.

Porting an algorithm to the Cell/B.E. architecture typically follows the scheme in figure 3.5 [9]. First, it is ported to run (sequentially) on the PPE. Then, by runtime profiling, the hotspots (or bottlenecks) of the algorithm are detected, i.e. the parts of the algorithm consuming most of the runtime. Each hotspot is parallelized and offloaded to the SPEs individually. The tuning of the SPE code, the synchronization and the data transfers follows.

To offload a function to the SPEs, it needs to be parallelizable. For this purpose, the workload usually is subdivided into smaller pieces, which can be processed independently. Sometimes this requires reorganizing the data structures if the original code was written without parallelization in mind. If data belonging to the same piece are located in contiguous memory regions, they can be transferred more efficiently between the local store and system memory. Furthermore, as usual in parallelization, one has to distinguish between shared data and private data. In the case of shared data, access by different threads needs to be synchronized.

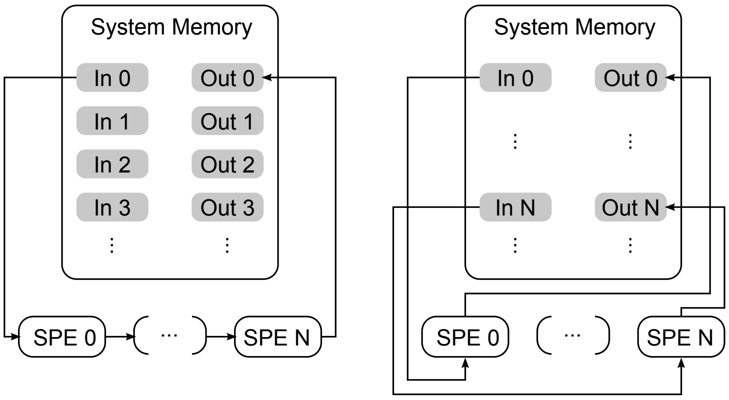


Fig. 3.6: The SPEs can be used in pipelined mode (left) or streaming mode (right).

The SPEs can either be used in pipelined or streaming mode (Figure 3.6). In pipelined mode, the offloaded function is split into pipeline stages. Each SPE executes a different kernel, corresponding to one pipeline stage. SPE0 processes the first piece and when finished hands it to SPE1 and starts working on the next piece. A piece is moved through the pipeline and the final result is stored to system memory by the last SPE in the chain. SPE-to-SPE DMA transfers can be used between the pipeline stages instead of storing the temporary results to system memory and loading them again. This is more efficient because the on-chip EIB has a much higher bandwidth and a much lower latency than external memory. Load balancing can be an issue if the time required for the different stages varies considerably.

In streaming mode, all the SPEs execute the same kernel on different data. The distribution of the pieces to the SPEs can be static. This is especially efficient if all the pieces take more or less the same time for processing. Otherwise, load imbalance can hamper performance and dynamic scheduling may be necessary, either managed by the PPE or by the SPEs collaboratively.

Regardless of the programming model, the main thread is running on a PPE. This thread creates subthreads running on the SPEs. This is similar to shared-memory programming with Pthreads. The `libspe2` library, which is part of the Cell/B.E. SDK, provides functions for SPE management. First, a context data structure needs to be created with the `spe_context_create` function. A context is a logical representation of an SPE and contains all its persistent data. The `spe_program_load` function loads an SPE program to the context. The SPE program is a separate program, having its own `main` function.

The `spe_context_run` function causes the loaded program to be executed. It is a synchronous function, meaning that it blocks the PPE thread calling it until the SPE thread returns. Therefore, the main PPE thread typically spawns one Pthread (running on the PPE) per SPE. This Pthread then takes care of the management of one SPE. When a context is not used any more, it can be destroyed with the `spe_context_destroy` function.

3.4.8 Rigid Registration

While not much is known about nonrigid registration on the Cell/B.E. platform, Ohara et al. [75] implemented a rigid registration algorithm for this processor. On a dual processor Cell/B.E. blade, they achieved a 11× performance improvement compared to a sequential `ITK` (Insight Toolkit, a open source collection of registration and segmentation algorithms) implementation running on a Xeon processor. Mutual information is used as the similarity metric. The algorithm uses subsampling, a common technique to accelerate rigid registration algorithms. It uses only 1% of the fixed image voxels, which is mainly possible because of the much smaller number of parameters compared to a nonrigid transformation model. The subset of fixed image voxels does not change between the iterations of the optimizer. In the first iteration, scattered (and thus inefficient) memory accesses are required to fetch the $2 \times 2 \times 2$ neighborhood required to interpolate the moving image at the mapped coordinates. After the first iteration, these voxels are stored together, such that they can be fetched again as a group in the next iteration, which is more efficient. For each fixed image voxel used for mutual information computation, there is one moving image group. However, if during the iterative optimization the transformation parameters change much, different moving image voxels may be required than in the first iteration. If this is the case, a scattered fetch of the new group is required and it is stored as a group again. To reduce these scattered fetches, a group of voxels is also used for extrapolation if the mapped coordinates are slightly outside of the region defined by this group.

3.5 GPU

3.5.1 Overview

Graphics processing units (GPUs) are another specialized platform. Maybe they offer the best computational power per price ratio nowadays. Although originally designed specifically for accelerating 3D graphics rendering, GPUs receive a growing attention outside of this domain. Researchers and developers are interested in harnessing the computation power of GPUs for other applications. As the GPGPU community (General-Purpose computation on Graphics Processing Units) is growing, GPUs are used to accelerate computation for more and more applications. A survey by Owens et al. [78] summarizes the principal hardware and software developments in the field of GPGPU.

While computational performance of CPUs grows at a rate of roughly 1.4 per year, the typical performance metrics for GPUs improve at higher rates between 1.7 and 2.3 yearly. Thus, GPUs are not only faster for certain applications already today, but their advantage compared to CPUs can even be expected to increase in the future. Both platforms base on the same manufacturing technology, which consequentially improves at the same rate for both. The computational performance of GPUs has always been based on data-level parallelism. Graphics processors have a large number of highly specialized processing units.

However, GPUs also have become more flexible over time. While early GPUs had fixed-function pipelines and the output was limited to 8-bit values per color-channel, modern GPUs feature programmable execution units and support single-precision floating-point data. New features are added to every major new generation of GPUs. But the power of GPUs still results from a highly specialized architecture and while they can be used today for many applications outside the originally narrow field for which they were designed, there remains a large class of applications for which they are not suitable and probably never will be. For example, GPUs lack efficient scatter operations. Writing a value to an indexed position of an array is not straightforward.

3.5.2 System Architecture

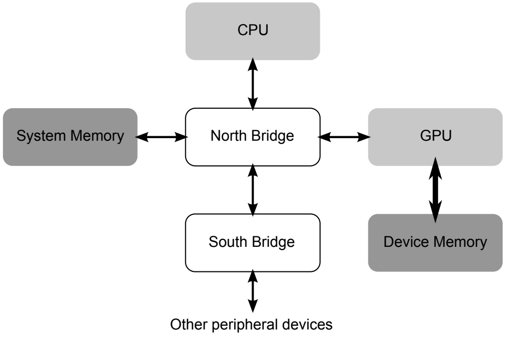


Fig. 3.7: System architecture with GPU.

In typical PC systems with GPUs, the CPU is connected to the main memory and the GPU by the North Bridge, or memory-controller hub-chip. On recent systems, a PCIe bus connects the North Bridge and the GPU. The GPU has access to some dedicated device memory. The bandwidth to device memory usually is much larger than the one of the PCIe bus. Data has to be transferred from system memory to device memory in order to provide fast access to programs running on the GPU. Typically, there is less device memory than system memory and when working with large data sets, it may not be possible to keep all the required data in device memory. If this is the case, the relatively slow PCIe bus may become a bottleneck because more data is moved between system and device memory.

3.5.3 Processor Architecture

The basic task of a 3D graphics system is to synthesize, or render, an image from a description of a scene described with triangles. More complicated shapes, like curved surfaces, are approximated with triangular patches. The input data provided to the GPU are the vertices of the scenery and texture, which is image data that is “draped” over the geometry.

Modern GPUs, except the very latest ones (discussed later), all have a similar structure (Figure 3.8). Kilgariff and Fernando give a detailed description of the GeForce 6 Series GPU architecture [51]. The processing is organized in a pipelined fashion. The pipeline stages have become more

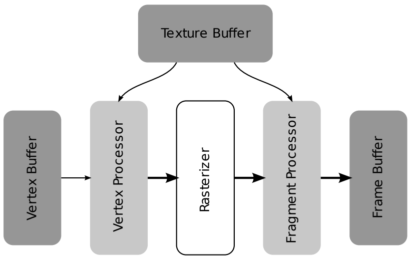


Fig. 3.8: The graphics pipeline of a modern GPU. The vertex and the fragment processor are programmable. Data is read from vertex and texture buffers. The result is written to the frame buffer. It it is an intermediate result, it may be necessary to copy the output to an input buffer (vertex or texture buffer) for further processing.

flexible over time, which has enabled more and more applications for which GPUs were originally not designed.

The vertex processor, or vertex shader, fetches data from the vertex buffer. It allows to apply per-vertex operations to each vertex in an object. After passing the vertex processor, vertices are grouped into primitives (points, lines or triangles). In more recent GPUs, the vertex processor can also access texture data.

For each primitive, the rasterizer calculates which pixels it covers. It outputs fragments, which can be interpreted as "candidate pixels". Further down the pipeline, fragments that are not visible, for example because they are occluded by other fragments, are discarded. Fragments that are not discarded will carry color information for the corresponding pixel. This color information is written to the render target, for example the frame buffer.

The fragment processor, or pixel shader, applies a shader program to each fragment independently. Its texture unit can work on texture data. Texture data can be seen as 2D or 3D arrays, which can be read at arbitrary coordinates, for example based on bi-linear or tri-linear interpolation. The fragment processor is the second programmable stage of the graphics pipeline.

GPUs draw their performance from multiple vertex processors and fragment processors working in parallel. The NVIDIA Geforce 7950 GX2, for example, has 16 vertex shaders and 48 pixel shaders.

Workload balancing is a possible problem associated with this pipeline design. If, for example, a simple scenery consisting of huge triangles is rendered, the vertex processors are almost idling, while the fragment processors are the bottleneck. On the other hand, if a very detailed scenery is rendered, the vertex processors become the bottleneck.

The introduction of unified shaders is a very recent step in the evolution of GPUs from highly specialized, hardwired processors to more and more flexible general purpose devices [63]. They were first available on the chip for the Xbox 360 gaming console and then brought to the PC market with the GeForce 8800 chip. These unified shaders are allocated in a grid and general enough to handle any of the shader workloads. Thus, vertices and fragments circulate through the grid instead of flowing through a pipeline with a fixed width per stage. The GeForce 8800 processors have up to 128 of these unified shaders, called stream processors.

3.5.4 Programming Model

As discussed in the preceding section, GPUs exploit parallelism to achieve their high performance. Furthermore, the architecture differs in many aspects from CPUs. Therefore, they require a different programming approach.

There are the traditional graphics APIs, such as the cross-platform OpenGL (Open Graphics Library) or Direct3D for Microsoft Windows platforms. High-level shading languages like Cg (or C for graphics) with its C-like syntax and GLSL (OpenGL Shading Language) also stick tightly to the GPU hardware design.

The general GPU programming approach is that the programmer identifies the data-parallel stages of the application and writes a shader kernel for each of these stages. To invoke the kernel, (vertex) input data are provided to the GPU. After a pass through the pipeline, the result data per fragment are written to texture memory, unless the final result has been generated. Complex algorithms may require dozens of passes through the pipeline.

For general purpose GPU programming, the graphic-centric nature of this programming model often is often a burden, especially if the application has nothing to do with rendering geometric primitives. GPGPU programming languages usually abstract the graphics specific terminology and see the GPU more as a streaming processor. Such programming environments include Close to metal by AMD/ATI (later called Stream SDK), CUDA (compute unified device architecture) by Nvidia and Stanford University's

BrookGPU. Especially CUDA boosted research and development in the GPGPU environment. OpenCL (open computing language) is an emerging framework for heterogeneous platforms consisting of CPUs, GPUs and possibly other processors. It defines a C-like language for writing kernels for OpenCL devices and APIs to set up and coordinate the heterogeneous platform.

3.5.5 Early Nonrigid Registration Work

The work of Soza et al. [90] presents early research on GPU-based acceleration of nonrigid registration. They use a transformation function, parametrized by displacement vectors located on a $5 \times 5 \times 5$ grid. The dense transformation field is obtained using Bézier functions. The multidimensional optimization problem is addressed with Powell's direction set method. Only one parameter is optimized at a time. The quality of the fit is assessed by a mutual-information-based similarity metric. The GPU is used to accelerate the transformation of the moving image, i.e. computing the dense transformation function and interpolating the moving image at the transformed coordinates. The algorithm is used to compensate brain shift between pre- and intraoperative MR images. The original images have a resolution of $256 \times 256 \times 112$ voxels. However, they are downsampled prior to registration in order to further speed up the process. On a commodity system with an AMD Athlon 1.2 GHz processor and a GeForce3 64MB graphics card, average runtimes around six to seven minutes are achieved.

3.5.6 Gradient Flow Algorithms

A fully GPU-based nonrigid registration algorithm was presented by Strzodka et al. [93] in 2004. A gradient flow registration algorithm [20] was implemented for the DirectX 9 graphics pipeline. This algorithm minimizes the SSD criterion. Similar to the Demons algorithm, the dense transformation field is updated iteratively by following a gradient descent step. For regularization, a smoothing filter is applied to the gradient before the step (in contrast to the Demons algorithm, were the smoothing filter is applied afterwards). The two dimensional fixed and moving images are represented as 2D textures. They are transferred to the graphics memory at the beginning. When the registration finishes, the result is sent directly from graphics memory to the screen. No further passing of image data between main and graphics memory are necessary. A C++ program running on the CPU configures the graphics pipeline for the different stages of the algorithm. It controls which vertex and fragment processor kernels to use and sets the pointers to the input data. For each part of the algorithm, a different kernel is required. These kernels are programmed in the high level graphics programming language Cg.

In this multi-iteration algorithm, result images from one step are the input of the next step. To pass these images from one iteration to the next one, a technique called "render to texture" is used. Specifically, Strzodka et al. use `pbuffers` for this purpose. They observed when switching the output buffer. They expect a significant performance improvement once a more efficient way to render to texture is supported by GPUs.

Registration time of approximately 3 seconds for 257^2 images and up to 10 seconds for 513^2 images is achieved. An NVIDIA GeForceFX 5800 Ultra chip is used for the experiments. This was a high-end GPU released in 2003. A pure CPU implementation is reported to be four times slower.

In 2006, Köhn et al. [53] extended the work by Strzodka et al. to three dimensions. Furthermore, they added a rigid transformation that precedes the nonrigid registration. The GPU is programmed in the OpenGL language GLSL. The `pbuffers` used by Strzodka et al. were replaced by Framebuffer objects, which are a faster alternative supported by the OpenGL 2.0 specification. On a GeForce 6800 GT graphics processor, the runtime of 2D nonrigid registration for an 256^2 image is reduced to 200 ms, which is more than 20× faster than on a CPU.

However, for 3D registration, a new bottleneck appeared. Rendering to a slice of a 3D volume is not supported directly and the output of the graphics pipeline needs to be copied before it can be used as an input for the next pass. Köhn et al. estimate that the resulting overhead amounts to 80%-90% of the total computation time in the 3D case. They report a runtime of 62.5 seconds for a $256 \times 256 \times 128$ image, expecting this number to drop to 9.2 seconds once the copy bottleneck is overcome.

3.5.7 Demons Algorithm

Several research groups worked independently on GPU implementations of the Demons algorithm and published their work in 2007 and 2008. In order to get an overall smooth mapping, the transformation field is regularized using a Gaussian smoothing filter after each iteration. This filtering step is computationally expensive. The different Demons algorithm implementations mainly differ on the smoothing method and the programming environments.

Sharp et al. [89] use the Brook programming language to implement their “streaming version” of the Demons algorithm. Separable Gaussian filters, implemented as convolution kernels, are used for the smoothing step. The algorithm is validated using CT data of a swine lung, which is warped by a known deformation. The performance of the GPU implementation, running on an NVIDIA 8800 GTS GPU, is compared to the sequential implementation running on an 2.8 GHz Intel Duo-Core processor. For 100 iterations with an $424 \times 180 \times 150$ image, a runtime of 13.41 seconds is achieved, which corresponds to a speedup of 45.51× compared to their CPU implementation.

Courty and Hellier [24] present Cg-like code fragments of their implementation. They map the volumetric image data to 2D textures, which they call flat 3D textures. This allows them to process the entire volume in one rendering pass. Furthermore, the GPU does not need to support render-to-3D-texture extensions. The Gaussian filtering is implemented as a separable recursive filter. It is approximated with 4^{th} order cosine-exponential functions. Two passes are required per direction, one for the causal part of the filter and another one for the anticausal part. To handle the causal and the anticausal part at the same time, two copies of the flat 3D texture, decomposed in positive and negative direction of the respective axis, are stored in the red and the green channel of the 2D texture. After each pass of the recursive filter, the image has to be reoriented to allow a sweep in the next direction. An advantage their algorithm is that its runtime does not depend on the standard deviation parameter σ.

For the verification of the algorithm, it was applied to 3D brain MR images on a system consisting of an Athlon XP 2500+ CPU and a Quadro FX 1400 graphics card. The original images were downsampled to 128^3 voxels because of the limited size (256 MB) of the GPU device memory. A runtime of 2.2 seconds per iteration was measured.

Muyan-Özçelik et al. [72] use the same approach to Gaussian smoothing as Sharp et al. do. However, their implementation is written in CUDA and they claim a 10% performance improvement on the same hardware compared to the Brook-based implementation. They state that CUDA specifically targets newer NVIDIA cards, such as the one used by Sharp et al., while Brook lacks support for some of the latest features. On an Nvidia

8800 GTS GPU, a runtime of 12.46 seconds was achieved for 100 iterations with an $424 \times 180 \times 150$ image. A large part of the speedup was provided by coalescing of the global memory accesses, which resulted in a 3.9× speedup. Simultaneous memory reads from multiple threads can be coalesced into a single memory transaction if certain alignment conditions are fulfilled [74]. They also compare their GPU implementation to a CPU implementation and observe a 55× speedup compared to a single-threaded and 35× compared to a multi-threaded implementation.

3.5.8 Finite Element Algorithm

Li et al. [59] present an OpenGL-based implementation of a finite element nonrigid registration algorithm. It is limited to 2D images and applied to motion tracking in cardiac MRI. The transformation function bases on a finite element model and interpolation with bi-cubic Bézier basis functions. SSD (sum of squared differences) is applied as similarity metric. The GPU is used to compute the transformed moving image. The gradient of the metric with respect to the parameters as well as the Hessian matrix[5] are calculated on the GPU and transferred to the CPU. A Levenberg-Marquardt[6] optimizer, running on the CPU, is applied to iteratively optimize the parameters of the finite element model. The authors plan to migrate to CUDA in the future. They expect that this would yield further performance improvements by allowing them to offload the optimizer to the GPU.

On an NVIDIA 7950 GTX graphics processor, registration of a region of interest (ROI) of 128^2 pixels takes 0.46 seconds. The same task is reported to take 2.12 seconds on an Intel dual-core 2.0 GHz processor, corresponding to a speedup of 4.6×.

3.5.9 Multimodal Registration

The nonrigid registration algorithms for GPUs discussed so far all target monomodal registration. An exception is the early work by Soza et al., but their implementation only offloads the transformation of the moving

---

[5] The Hessian matrix is the square matrix of the second order partial derivatives of a function (in this case the metric).

[6] The Levenberg-Marquardt algorithm iteratively solves the problem of minimizing a function. It requires the computation of the gradient and the Hessian matrix of this function.

image to the GPU. An implementation published by Vetter et al. in 2007 [102] uses mutual information as the similarity metric and therefore allows multimodal registration. However, the implementation is limited to 2D images. The GPU is programmed with OpenGL and GLSL. The similarity metric has two terms in addition to mutual information. It also calculates the Kullback-Leibler divergence between a learned and the observed joint probability density function. The Kullback-Leibler divergence is a measure for the difference between two probability density functions. For the registration algorithm, this means that transformations yielding a joint probability density function similar to the learned one are preferred. This method provides context-specific information, which is missing when using mutual information alone. Furthermore, a regularization term is used to favor smooth transformation fields. The optimal solution is searched with a gradient descent optimizer.

The authors report a speedup of 5.9× compared to a CPU implementation. For a 512^2 pixel image, the runtime per iteration is reduced from 22.95 to 3.9 seconds.

3.5.10 Conclusions

GPUs are receiving more and more attention as cost-efficient high-performance platforms for parallelizable algorithms. Recently, this has resulted in a number of publications on GPU implementations of nonrigid registration algorithms. Some research groups even reported registration times in the order of seconds. However, most implementations only support monomodal registration. The only multimodal nonrigid registration algorithm available for GPUs is limited to 2D images. Although the authors do not mention the number of iterations typically required to solve a registration problem, the fact that an iteration takes 3.9 seconds for an 512^2 pixel image suggests a total registration time in the order of one to a few minutes.

Algorithms usually have to be re-designed to be mapped to the specialized processing flow of GPUs. Programming complex algorithms for these architectures is challenging, which may be a reason for the limitations of the available methods. However, emerging higher level programming languages have recently improved the programmability of graphics processors. Furthermore, new generations of GPUs usually also have additional features, increasing their capabilities for general purpose processing. These trends may enable the implementation of a broader spectrum of algorithms on

GPUs, which would be desirable considering the pace with which computational capabilities of graphics processors grow.

3.6 FPGA

3.6.1 Architecture

The Cell/B.E. processor and GPUs are processor architectures specialized on computationally intensive tasks. A different platform which continues to attract researchers in the accelerator environment are field programmable gate arrays (FPGAs). They are reconfigurable devices intending to fill the gap between software-programmed microprocessors and hardwired Application Specific Integrated Circuits (ASICs) [23]. While ASICs are designed to do one specific task with a very high performance, microprocessors can be applied to a broad range of tasks, but possibly with lower performance than a specialized chip. FPGAs contain a large number of computational elements (or logic blocks). Their functionality is programmed through configuration bits. Furthermore, the connections between the logic blocks are also programmable. This allows to map custom digital circuits to the FPGA.

There are different ways to attach an FPGA accelerator to a host processor. There are FPGA boards, which are usually connected to the host processor through a PCI (Peripheral Component Interconnect) bus. Like GPUs, the FPGA typically also has its own device memory. More recently, some FPGA accelerators can also be plugged directly into a normal processor socket, which allows to couple processor and accelerator more tightly. The accelerators include an on-chip memory controller, which enables communication through shared memory [107].

3.6.2 Volume-Subdivision Algorithm

An FPGA-based implementation of a volume-subdivision registration algorithm was reported to outperform a (single-threaded) software implementation by a factor 30× [27]. The registration time for 256^3 images is reduced to around 6 minutes. The iterative optimizer running on the host CPU provides candidate transformations to the accelerator, which is implemented on a PCI attached prototyping board with dedicated memory. The accelerator calculates a mutual-information-based similarity metric and provides it to the optimizer. This step requires the calculation of the (rigid) transformation within the subvolume, partial volume interpolation at the transformed coordinates and updating the joint histogram of the two images. Finally, the entropies and then the mutual information are calculated. The necessary calculation of logarithms is implemented using lookup tables.

3.7 Conclusions

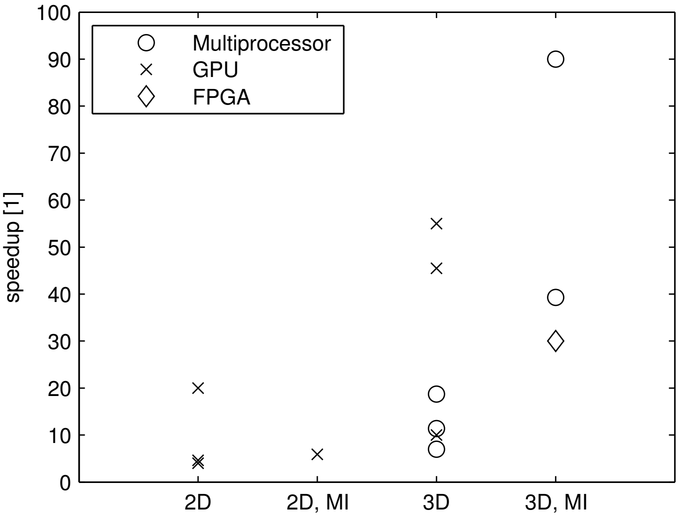


Fig. 3.9: Speedup comparison of different nonrigid registration algorithms. They are categorized with respect to the number of image dimensions (two or three) and similarity metric. MI means mutual-information-based. The other algorithms are not suited for multimodal registration.

In figure 3.9 we plot the speedup of the nonrigid registration algorithms discussed in the preceding section where the authors provide this information. Typically, the speedup is relative to a sequential version of the respective algorithm running on one CPU. We see that for 3D registration higher speedups are achieved than for 2D. This may well be because acceleration of 3D registration provides more incentive. 2D registration times of sequential algorithms are already relatively low at least for the image sizes used in the cited studies. The highest speedups are achieved for 3D mutual-information-based registration, but the utilized multiprocessor systems are relatively large and expensive. There is a much smaller FPGA-based solution providing significant speedups for the same class of algorithms. For monomodal registration, GPUs seem to be a cost-efficient platform providing very good performance.

4

Implemented Registration Algorithm

4.1 Basic Features

In the preceding chapters, different registration techniques and accelerated registration algorithms were discussed. Many of the fastest implementations use an intensity-based similarity metric (rather than to geometric features, such as landmarks) and a generic transformation model (in contrast to a problem-specific model of the anatomy). The choice of the similarity metric and the transformation model may be the most important design decision. Our goal is a fast 3D nonrigid registration algorithm on a relatively low-cost platform. The performance is achieved by exploitation of parallelism. Furthermore, the algorithm should be suitable for a broad range of applications, including multimodal problems.

Mutual information is chosen as the similarity metric because it is generally applicable. Numerous clinical applications have been reported. Although first introduced for the purpose of multimodal registration, it was also successfully applied to monomodal problems [81]. It usually can be used without preprocessing the images or user interactions, such as parameter tuning or initialization.

The nonrigid transformation will be modeled with B-splines, a technique which has been applied to a variety of anatomical regions, such as the brain

[45], the chest [67], the heart [31, 70, 58], the liver [84] and the breast [86]. It uses a relatively compact representation of the transformation function: the B-spline coefficients located at the transformation grid knots. This may result in less communication overhead than the direct use of a dense transformation field like for the Demons algorithm. Moreover, the computation of the transformation at an arbitrary point is efficient. It does not depend on the complexity of the model because only a fixed number of B-spline coefficients adjacent to the point are required to compute the transformation function.

4.2 B-Spline Transformation Model

The goal of image registration is to find the transformation function $\mathcal{T}$, which maps points with coordinates $\mathbf{x}_{fix}$ in the fixed image to their corresponding point $\mathbf{x}_{mov}$ in the moving image:

$$\mathbf{x}_{mov} = \mathcal{T}\left(\mathbf{x}_{fix}\,|\boldsymbol{\mu}\right), \tag{4.1}$$

where $\boldsymbol{\mu}$ is a set of transformation parameters. This allows warping of the moving image f_{mov},

$$f'_{mov}\left(\mathbf{x}\right) = f_{mov}\left(\mathcal{T}\left(\mathbf{x}\,|\boldsymbol{\mu}\right)\right), \tag{4.2}$$

such that the points (i.e., pixels for 2D and voxels for 3D images) of the transformed moving image f'_{mov} should be comparable (if not identical) to the points at the same coordinates in the fixed image f_{fix}.

We use a well-known transformation model, which is composed of a rigid (or global) and a nonrigid (or local) part [86].

$$\mathcal{T}\left(\mathbf{x}\right) = \mathbf{t} + \mathbf{A}\mathbf{x} + \mathcal{T}_{nonrigid}\left(\mathbf{x}\right) \tag{4.3}$$

The rigid transformation is defined by a translation vector $\mathbf{t}$ and a matrix $\mathbf{A}$. The nonrigid transformation $\mathcal{T}_{nonrigid}$ is based on B-splines. Usually, the parameters of the rigid transformation are found prior to nonrigid registration. Rigid registration is less time consuming than nonrigid registration and very fast implementations exist (e.g. by Ohara et al. [75]). Therefore, we assume that the parameters of the rigid transformation were already found and treat them as constants throughout the nonrigid registration. Alternatively, the parameters can be initialized, for example to centrically align the images and compensate differences in the sampling rate.

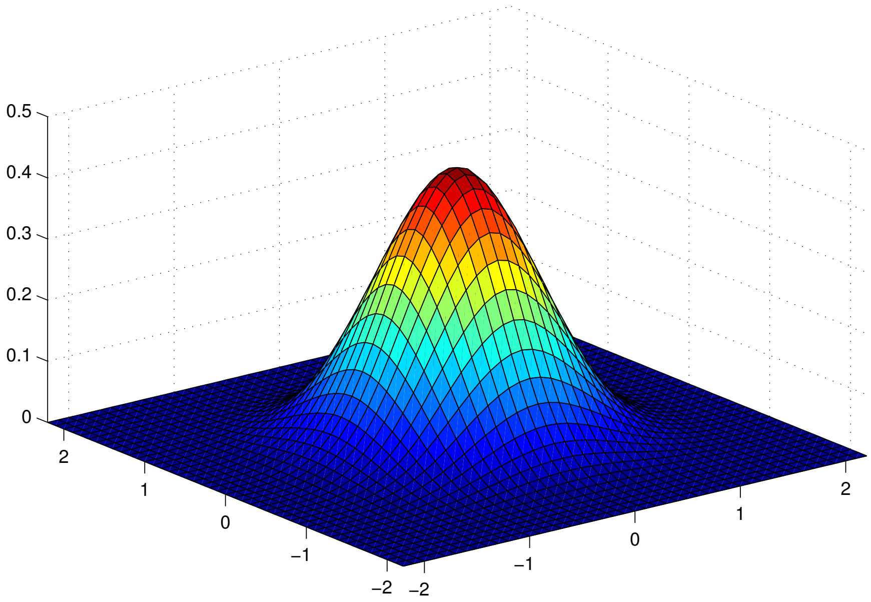


Fig. 4.1: 2D cubic B-spline base function

The cubic B-spline base function is

$$\beta_3(x) = \begin{cases} 0 & 2 \le |x| \\ \frac{1}{6}(2-|x|)^3 & 1 \le |x| < 2 \\ \frac{1}{6}\left(4-6x^2+3|x|^3\right) & 0 \le |x| < 1 \end{cases} \tag{4.4}$$

For N dimensions, $\beta_3(\mathbf{x})$ is a product of N 1D cubic B-spline functions.

$$\beta_3(\mathbf{x}) = \prod_{k=1}^{N} \beta_3(x_k) \tag{4.5}$$

Figure 4.1 shows the two-dimensional cubic B-spline base function. The nonrigid mapping is modeled by a number of shifted and scaled cubic B-splines, which are centered on the knots of a regular grid. The function is defined as

$$\mathcal{T}_{nonrigid}(\mathbf{x}) = \sum_{\mathbf{j}} \mathbf{c_j}\beta_3\left(\frac{\mathbf{x}}{h} - \mathbf{j}\right) \tag{4.6}$$

where $\mathbf{j}$ are the grid indices for the parameters $\mathbf{c}$ (the B-spline coefficients, for an N dimensional field, $\mathbf{c_j}$ has N components) and h specifies the grid spacing, which is not necessarily the same in all directions.

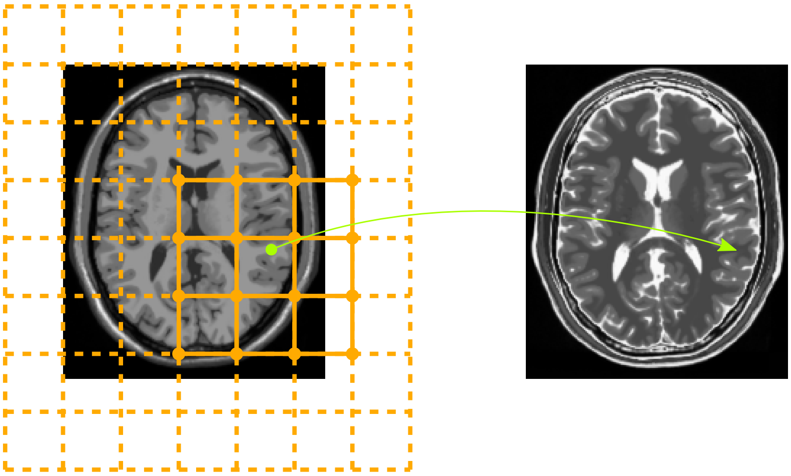


Fig. 4.2: A mesh of control points is laid over the fixed image. The transformation is modeled with B-spline coefficients. They define the scaling of cubic B-spline base functions placed at the control points. In the 2D case, a neighborhood of 4×4 control points has influence on the transformation function at a specific point.

In order to evaluate (4.6) at an arbitrary coordinate $\mathbf{x}$, only 4^N indices $\mathbf{j}$ result in nonzero weights $\beta_3\left(\frac{\mathbf{x}}{h} - \mathbf{j}\right)$. This is because the cubic B-spline function evaluates to nonzero values only in the range $]-2;2[$. Therefore, only the respective coefficients $\mathbf{c_j}$ are required for the calculation (Figure 4.2).

The entire set of parameters is therefore $\boldsymbol{\mu} = \{\mathbf{t}, \mathbf{A}, \mathbf{c}\}$, but as mentioned earlier, the translation vector $\mathbf{t}$ and the matrix $\mathbf{A}$ are obtained from a linear registration and are treated as constants.

4.3 Mutual Information

The negative mutual information between the fixed and the transformed moving image (with transformation parameters $\boldsymbol{\mu}$) can be computed like

$$S(\boldsymbol{\mu}) = -\sum_{\iota}\sum_{\kappa} p(\iota, \kappa; \boldsymbol{\mu}) \log\left(\frac{p(\iota, \kappa; \boldsymbol{\mu})}{p_{fix}(\iota; \boldsymbol{\mu})\, p_{mov}(\kappa; \boldsymbol{\mu})}\right), \tag{4.7}$$

with fixed image intensities ι and moving image intensities κ. This requires estimates of the (marginal) probability density functions $p_{fix}\left(\iota;\boldsymbol{\mu}\right)$ and $p_{mov}\left(\kappa;\boldsymbol{\mu}\right)$ and of the joint probability density function $p\left(\iota,\kappa;\boldsymbol{\mu}\right)$ in the overlapping region of the fixed and the transformed moving image. A straightforward approach is the computation of the joint and marginal histograms of intensities. Normalization of the histograms yields the respective probability density functions. Usually, there is not one histogram bin per possible intensity value but a range of intensity values corresponds to one single histogram bin. If L_{fix} and L_{mov} are uniformly sized bins with sizes l_{fix} and l_{mov}, intensities are linearly scaled to fall into a valid bin. The numbers of bins L_{fix} and L_{mov} are parameters of the algorithm and the respective bin sizes are computed from the ranges of intensities $[i_{fix,min}; i_{fix,max}]$ and $[i_{mov,min}; i_{mov,max}]$, and the values of L_{fix} and L_{mov} like

$$l_{fix} = \frac{i_{fix,max} - i_{fix,min}}{L_{fix}} \tag{4.8}$$

$$l_{mov} = \frac{i_{mov,max} - i_{mov,min}}{L_{mov}}. \tag{4.9}$$

The bins are indexed by integer values $\iota (0 \leq \iota < L_{fix})$ and $\kappa (0 \leq \kappa < L_{mov})$. The fixed image intensity $f_{fix}\left(\mathbf{x}\right)$ is mapped to the bin

$$\iota = \left\lfloor \frac{f_{fix}\left(\mathbf{x}\right) - i_{fix,min}}{l_{fix}} \right\rfloor. \tag{4.10}$$

Parzen windowing is an alternative approach used for probability density function estimation [96]. It also uses binning, but generates continuous functions. Choosing an appropriate Parzen window function provides differentiable probability density function estimates, which will prove to be an advantage. As proposed by Mattes et al. [67], we choose a cubic B-spline for the moving image intensities and a zero-order B-spline

$$\beta_0\left(x\right) = \begin{cases} 1 & 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

for the fixed image intensities. This yields the joint histogram estimation

$$\begin{aligned} p\left(\iota,\kappa\left|\boldsymbol{\mu}\right.\right) =& \alpha \sum_{\mathbf{x}\in V} \beta_0 \left(\iota - \frac{f_{fix}\left(\mathbf{x}\right) - i_{fix,min}}{l_{fix}}\right) \\ & \times \beta_3 \left(\kappa - \frac{f_{mov}\left(\mathcal{T}\left(\mathbf{x}\left|\boldsymbol{\mu}\right.\right)\right) - i_{mov,min}}{l_{mov}}\right). \end{aligned} \tag{4.12}$$

The set V is the overlap region of the fixed and the transformed moving image and α is the normalization factor, such that $\sum_{\iota,\kappa} p\left(\iota, \kappa \,|\boldsymbol{\mu}\right) = 1$. Because of the limited support of the Parzen window functions, one $\mathbf{x}$ contributes to four bins of the histogram.

The marginal probability density functions can be obtained like

$$p_{fix}\left(\iota\right) = \sum_{\kappa} p\left(\iota, \kappa \,|\boldsymbol{\mu}\right) \tag{4.13}$$

and

$$p_{mov}\left(\kappa\right) = \sum_{\iota} p\left(\iota, \kappa \,|\boldsymbol{\mu}\right). \tag{4.14}$$

4.4 Moving Image Model

The evaluation of (4.12) requires interpolation of the moving image at arbitrary coordinates: $f_{mov}\left(\mathcal{T}\left(\mathbf{x}\,|\boldsymbol{\mu}\right)\right)$. Different interpolation schemes are possible and the choice is usually a trade-off between accuracy and speed. Like Thévenaz and Unser [97], and Mattes et al. [67], we choose cubic B-splines for this task and the moving image intensity at arbitrary coordinates is interpolated like

$$f_{mov}\left(\mathbf{x}\right) = \sum_{\mathbf{j}} \mathbf{b_j} \beta_3\left(\mathbf{x} - \mathbf{j}\right). \tag{4.15}$$

The B-spline coefficients $\mathbf{b_j}$ are obtained by recursive filtering of the moving image samples [100, 101]. They are located on a regular mesh, which coincides with the position of the moving image voxels. Again, the cubic B-spline weights only evaluate to nonzero values in a neighborhood of control points.

The choice of the interpolation method is related to the chosen optimization strategy (see following section), which requires the computation of the gradient of the moving image. The additional complexity of cubic B-spline interpolation (for example compared to trilinear interpolation) is justified by the advantages in achieving better registration accuracy and converging in fewer iterations because of the better estimation of the image gradient [56]. Pluim et al. [81] and Hill et al. [41] compare different interpolation methods and associated problems.

4.5 Optimization Strategy

The optimal transformation parameters are found as

$$\mathbf{c}^* = \underset{\mathbf{c}}{\operatorname{argmin}}\, S\left(\mathbf{t}, \mathbf{A}, \mathbf{c}\right). \tag{4.16}$$

We can write the parameters $\mathbf{c}$ (the B-spline coefficients of the transformation) as a sequential array $(c_1, c_2, \ldots c_n)$. In 3D, there are three coefficients per grid knot. We apply a gradient descent optimizer with feedback step adjustment [57] to iteratively solve the minimization problem. In each iteration, a new set of coefficients $\mathbf{c}^{n+1}$ is calculated based on the old set $\mathbf{c}^n$ according to

$$\mathbf{c}^{n+1} = \mathbf{c}^n + \lambda \nabla S. \tag{4.17}$$

This optimizer requires the calculation of the gradient of the metric with respect to the transformation coefficients, which is given as

$$\nabla S = \left[\frac{\partial S}{\partial c_1}, \frac{\partial S}{\partial c_2}, \ldots, \frac{\partial S}{\partial c_n}\right] \tag{4.18}$$

The step size λ is updated after each gradient descent step, depending on the success of the step. A step is successful if the condition $S(\mathbf{c}^{n+1}) \leq S(\mathbf{c}^n)$ is met. After a successful step, λ is doubled. After an unsuccessful step, λ is divided by ten and the new set of parameters is discarded ($\mathbf{c}^{n+1} = \mathbf{c}^n$).

The chosen representations of the transformation, the probability density functions and the moving image, allow computing the gradient in closed form. The derivative of S with respect to a transformation coefficient c_i is

$$\frac{\partial S(\mathbf{c})}{\partial c_i} = -\sum_{\iota}\sum_{\kappa} \frac{\partial p\left(\iota, \kappa; \mathbf{c}\right)}{\partial c_i} \log\left(\frac{p\left(\iota, \kappa; \mathbf{c}\right)}{p_{mov}\left(\kappa; \mathbf{c}\right)}\right). \tag{4.19}$$

This step is described in detail in [97] and bases on the assumption that the marginal probability density function of the fixed image p_{fix} is independent of the transformation parameters (the fixed image is not deformed). The joint probability density function is calculated over the region of overlap V in (4.12) and V may change with $\mathbf{c}$. This and the fact that p_{fix} is calculated from p in (4.13) seems to imply that p_{fix} depends on $\mathbf{c}$. However, due to the discrete nature of V, an infinitesimal change of a parameter c_i does not result in an infinitesimal change of the overlap region. Therefore $\mathrm{d}V/\mathrm{d}c_i$ is zero almost everywhere.

The derivative of the joint probability density function with respect to the parameter c_i is obtained by multiple application of the chain rule:

$$\begin{aligned}\frac{\partial p\left(\iota,\kappa;\mathbf{c}\right)}{\partial c_i} =&\alpha\sum_{\mathbf{x}\in V}\beta_0\left(\iota-\frac{f_{fix}\left(\mathbf{x}\right)-i_{fix,min}}{l_{fix}}\right)\\ &\times\left.\frac{\beta_3\left(\tau\right)}{\partial\tau}\right|_{\tau=\kappa-\frac{f_{mov}(\mathcal{T}(\mathbf{x}|\mathbf{c}))-i_{mov,min}}{l_{mov}}}\\ &\times\left(-\left.\frac{\mathrm{d}f_{mov}(\boldsymbol{\xi})}{\mathrm{d}\boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=\mathcal{T}(\mathbf{x};\mathbf{c})}\right)^{\mathrm{T}}\cdot\frac{\partial}{\partial c_i}\mathcal{T}\left(\mathbf{x};\mathbf{c}\right),\end{aligned}\tag{4.20}$$

where

$$\left(\left.\frac{\mathrm{d}f_{mov}(\boldsymbol{\xi})}{\mathrm{d}\boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=\mathcal{T}(\mathbf{x};\mathbf{c})}\right)^{\mathrm{T}}\tag{4.21}$$

is the gradient of the moving image, which is continuous due to the high order interpolation. The term

$$\frac{\partial}{\partial c_i}\mathcal{T}\left(\mathbf{x};\mathbf{c}\right)\tag{4.22}$$

depends on geometry only. It is the variation of the transformation function depending on the variation of the parameter c_i. Because of the limited support of the B-spline basis function, (4.22) results in a vector with all components zero for $\mathbf{x}$ which are not in the neighborhood N_{c_i} of the grid knot corresponding to c_i. Within the support region, the resulting vector has exactly one nonzero component.

The sum in (4.20) can be calculated over a subset of V. If we combine (4.19) and (4.20) after reordering, we get

$$\begin{aligned}\frac{\partial S(\mathbf{c})}{\partial c_i} =&\alpha\sum_{\mathbf{x}\in N_{c_i}}\sum_{\iota}\sum_{\kappa}\log\left(\frac{p\left(\iota,\kappa;\mathbf{c}\right)}{p_{mov}\left(\kappa;\mathbf{c}\right)}\right)\\ &\beta_0\left(\iota-\frac{f_{fix}\left(\mathbf{x}\right)-i_{fix,min}}{l_{fix}}\right)\\ &\times\left.\frac{\beta_3\left(\tau\right)}{\partial\tau}\right|_{\tau=\kappa-\frac{f_{mov}(\mathcal{T}(\mathbf{x}|\mathbf{c}))-i_{mov,min}}{l_{mov}}}\\ &\times\left(\left.\frac{\mathrm{d}f_{mov}(\boldsymbol{\xi})}{\mathrm{d}\boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=\mathcal{T}(\mathbf{x};\mathbf{c})}\right)^{\mathrm{T}}\cdot\frac{\partial}{\partial c_i}\mathcal{T}\left(\mathbf{x};\mathbf{c}\right).\end{aligned}\tag{4.23}$$

For each $\mathbf{x}$, the term

$$\beta_0 \left(\iota - \frac{f_{fix}\left(\mathbf{x}\right) - i_{fix,min}}{l_{fix}} \right) \tag{4.24}$$

is nonzero for exactly one ι only. For this $\iota_0\left(\mathbf{x}\right)$, it evaluates to 1. Furthermore, the term

$$\left. \frac{\beta_3\left(\tau\right)}{\partial \tau} \right|_{\tau = \kappa - \frac{f_{mov}\left(\mathcal{T}\left(\mathbf{x}|\mathbf{c}\right)\right) - i_{mov,min}}{l_{mov}}} \tag{4.25}$$

is nonzero only for four adjacent κ, starting with $\kappa_0\left(\mathbf{x}\right)$. Therefore, (4.23) can be written as

$$\begin{aligned} \frac{\partial S(\mathbf{c})}{\partial c_i} =& \alpha \sum_{\mathbf{x} \in N_{c_i}} \sum_{\kappa = \kappa_0(\mathbf{x})}^{\kappa_0(\mathbf{x})+3} \log \left(\frac{p\left(\iota_0\left(\mathbf{x}\right), \kappa; \mathbf{c}\right)}{p_{mov}\left(\kappa; \mathbf{c}\right)} \right) \\ & \times \left. \frac{\beta_3\left(\tau\right)}{\partial \tau} \right|_{\tau = \kappa - \frac{f_{mov}\left(\mathcal{T}\left(\mathbf{x}|\mathbf{c}\right)\right) - i_{mov,min}}{l_{mov}}} \\ & \times \left(\left. \frac{\mathrm{d} f_{mov}(\boldsymbol{\xi})}{\mathrm{d}\boldsymbol{\xi}} \right|_{\boldsymbol{\xi} = \mathcal{T}(\mathbf{x};\mathbf{c})} \right)^{\mathrm{T}} \cdot \frac{\partial}{\partial c_i} \mathcal{T}\left(\mathbf{x}; \mathbf{c}\right), \end{aligned} \tag{4.26}$$

The values

$$\log \left(\frac{p\left(\iota, \kappa; \mathbf{c}\right)}{p_{mov}\left(\kappa; \mathbf{c}\right)} \right) \tag{4.27}$$

can be precomputed.

A multiresolution approach may increase robustness and speed [57], meaning that the original images are first downsampled and pyramids of the images of decreasing resolutions are built. The registration starts at the lowest image resolution and then successively moves down the pyramid until the desired resolution has been reached. The grid of control points gets refined at transitions between pyramid levels and initialized with the transformation found in the preceding level.

5

Parallel Algorithm Design

5.1 Problem Analysis

5.1.1 The Algorithm

The sequential nonrigid registration algorithm described in the preceding chapter follows the steps listed in listing 5.1.

Listing 5.1: Algorithm pseudo-code

```
for each iteration of the gradient descent optimizer
  iterate over all fixed image points // loop A1
    evaluate transformation function
  iterate over all fixed image points // loop A2
    interpolate moving image intensity at respective coordinates
  iterate over all fixed image points // loop A3
    get fixed image intensity
    get moving image intensity
    add entry to joint histogram
  calculate joint and marginal probability density functions
```

```
11  calculate the mutual information
12  precompute pdf ratios
13  iterate over all fixed image points // loop A4
14    evaluate transformation function
15  iterate over all fixed image points // loop A5
16    interpolate moving image intensity and its
         gradient
17  iterate over all transformation grid knots //
       loop A6
18    iterate over all points in the neighborhood
19      get fixed image intensity
20      get moving image intensity and its gradient
21      get respective pdf ratio
22      calculate the gradient of the metric at this
           point and add it to the total gradient of
           the respective grid knot
23  calculate the new set of coefficients  //
       gradient descent (gd)
```

In line 11, the mutual information is calculated. If we are not in the first iteration and the metric calculated during the preceding iteration was better, this means that the preceding gradient descent step (line 23) resulted in an inferior set of coefficients. This set is discarded and the last successful set is reloaded. Therefore, the gradient does not have to be recalculated (loops A4 to A6) but the old gradient can be used again with a different step size of the gradient descent optimizer. The pdf ratios in line 12 are the values (4.27), which are precomputed.

Figure 5.1 shows a producer-consumer diagram, which indicates which phases of the algorithm produce what data and where these data are consumed. Our goal is to accelerate this algorithm. In a first step, optimizations of the (sequential) algorithm are discussed and remaining bottlenecks identified. In a second step, the algorithm is parallelized.

5.1.2 Performance Bottlenecks

In order to identify the performance bottlenecks, we profiled 30 iterations of the sequential registration algorithm on a standard processor using images with $181 \times 217 \times 181$ voxels (isotropic spacing of 1 mm) and a transformation grid spacing of 16 mm. Before we start looking into the parallelization of the algorithm, we try to find ways to optimize the sequential algorithm.

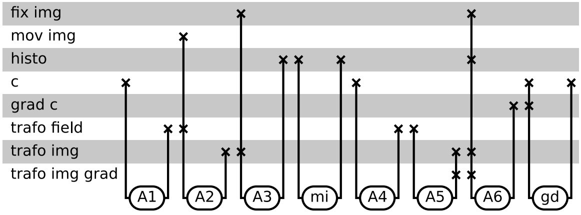


Fig. 5.1: The producer-consumer diagram shows in which phases (blobs at the bottom) of the algorithm data structures (rows) are read or written. It shows, from left to right, the flow during one iteration. In phase A1, for example, the coefficients (c) are read and the dense transformation field is written (trafo field). The other important data structures are the fixed image (fix img), the interpolation coefficients of the moving image (mov img), the histogram (histo), the gradient of the metric with respect to the transformation coefficients (grad c), the transformed moving image (trafo img) and the gradient of the transformed moving image (trafo img grad). The phase "mi" combines lines 10 to 12 of the pseudo-code. Based on the histogram it calculates the mutual information and stores the pdf ratios (4.27) to the histogram data structure. These values are used in phase A6.

With a system profiler (Oprofile, see [4, 21]), it is possible to measure how much CPU time is attributed to each line of source code. For our workload, it reveals that more than 70% are spent with B-spline computations of the form

$$f = c\beta_x\beta_y\beta_z, \tag{5.1}$$

and computation of the sums of these products of B-spline coefficients c and weights β.

5.2 Optimizations

5.2.1 B-Spline Interpolation

We have seen that a large part of the computation time stems from the evaluation of the cubic B-spline weights and their multiplication with the B-spline coefficients when computing the transformation and moving-image interpolation. Because of the limited support of the cubic B-splines, the evaluation in the 1D case is a weighted sum of four coefficients. In the

3D case we need to work on a $4 \times 4 \times 4$ neighborhood of $\mathbf{x}$ to evaluate $T_{nonrigid}(\mathbf{x})$ or $f_{mov}(\mathbf{x})$. The basic 3D B-spline formula is

$$f(\mathbf{x}) = \sum_{i,j,k=0..3} c_{i,j,k} \beta_{x,i} \beta_{y,j} \beta_{z,k}, \tag{5.2}$$

where f is one component of the transformation function or the moving image intensity, $c_{i,j,k}$ are the coefficients with nonzero weight and the β are the B-spline weights.

The computation of each of the three components of the transformation vector is carried out independently, using the same weights β but different coefficients. The computation of the gradient (4.26) also involves multiplication with B-spline weights, because of its last factor $\frac{\partial}{\partial c}\mathcal{T}(\mathbf{x})$.

5.2.2 Use of Lookup Tables

In the case of the transformation function, we constrained the grid point spacing to be an integer multiple of the voxel spacing. As shown by Kybic and Unser [57], this allows the use of a lookup table for the B-spline weights instead of their explicit calculation at runtime without any loss of accuracy. Therefore, all the calculations of B-spline weights β are related to the moving image interpolation. Instead of using the exactly mapped coordinates for this interpolation, we can round the coordinates to a subgrid, for example with a spacing of 1/64 of the voxel size. A lookup table of the B-spline weights at the subgrid control points can then be calculated off-line and used during the registration. Such lookup tables are relatively small: an image subgrid table as defined before has 64 entries each containing four single-precision float coefficients $\{\beta_{x,0}, \beta_{x,1}, \beta_{x,2}, \beta_{x,3}\}$ $(\beta_{x,0} = \beta_{y,0} = \beta_{z,0}, \ldots)$ and therefore occupies 1kByte.

5.2.3 Vectorized B-Spline Interpolation

Many processor architectures feature some sort of SIMD (single instruction multiple data) extension. This technique, also called vector processing, provides data level parallelism by allowing to perform an instruction, for example a multiplication, on multiple data values at the same time. SSE (Streaming SIMD Extensions) for x86 processors or AltiVec for the Power architecture work on 128-bit registers, allowing for example to multiply in one operation two vectors consisting of four single precision float values each. The vector size is possibly a good match for cubic B-splines, which work on a $4 \times 4 \times 4$ neighborhood.

When rearranging (5.2), we get

$$f(\mathbf{x}) = \sum_{i=0..3} \beta_{x,i} \sum_{j=0..3} \beta_{y,j} \sum_{k=0..3} c_{i,j,k} \beta_{z,k} \tag{5.3}$$

A scheme to vectorize this calculation is illustrated in figure 5.2. We have 16 vectors of four B-spline coefficients each (arranged like the blue lines). We multiply each of the vectors with the correct β_z value (they are arranged as vectors in the lookup table) and added in groups of four based on their y index yielding four vectors (green). They are multiplied with the β_y weights and added to obtain the orange vector. The scalar result is found as the sum of the four components of the product of the orange vector with the β_x weights.

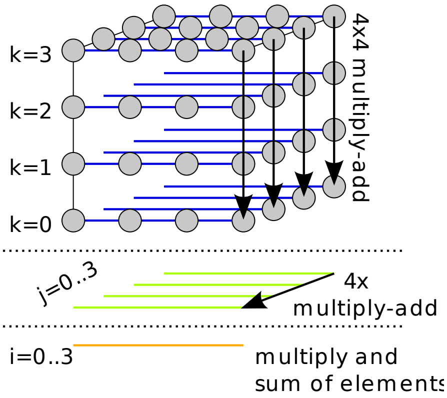


Fig. 5.2: Vectorized B-spline interpolation. Each colored line represents a SIMD vector containing four B-spline coefficients. For the vectorized spline evaluation, each of the 16 blue vectors is first multiplied with a vector containing the corresponding weights β_z. The results are added in groups of four to obtain the four green vectors. Vectors containing coefficients at the same y coordinate are in the same group. The green vectors are then multiplied with the corresponding weights β_y and added to obtain the orange vector. This vector is multiplied with the weights β_x and the four components of the result are added to obtain the final result of the interpolation.

5.2.4 Optimized Transformation Field Evaluation

Let us assume that the transformation grid spacing is 16 voxel units in all directions. It is efficient to calculate $T_{nonrigid}(\mathbf{x})$ together for groups of fixed image points that depend on the same B-spline coefficients, in our example $16^3 = 4096$ points. All the points of this group with the same z-coordinate (sub-groups of $16^2 = 256$ points) share the first step of the computation, the reduction from 16 blue to four green vectors, because the same weights β_z are used. If they additionally have the same y-coordinates (sub-groups of 16 points), they even share both the first and second step up to the reduction to one orange vector. If each point is processed individually, $16^3 \times (16 + 4 + 1) = 86016$ vector multiplications are required for the entire group. With grouping, the number of vector multiplications is reduced to only $16 \times 16 + 16^2 \times 4 + 16^3 \times 1 = 5376$.

A common overhead of SIMD parallelism is that the data has to be loaded to the vector processing registers. In the case of the Cell/B.E. processor, for example, the vector data can only be loaded efficiently from memory if the elements are stored at continuous memory locations, starting at a 16-Byte-aligned address. If the vector data are scattered in memory or not properly aligned, they have to be loaded using multiple load operations and further processing is required to assemble the vectors from the loaded data. We will see in section 6.1.7 that this can severely impact performance. While the weights in the lookup tables can be stored as aligned vectors, this is not possible for the transformation coefficients. However, if the fixed image points are processed group-wise, the coefficients have to be loaded and assembled only once for the entire group, which makes this overhead negligible compared to the actual computation.

5.2.5 Optimized Image Interpolation

For the moving image interpolation, a similar grouping is impossible because it is sampled irregularly (the transformed fixed image coordinates are not on a regular grid) and also not many points depend on the same moving image coefficients. But we can efficiently combine the calculation of the image intensity and its gradient (which has three components). The gradient in x-direction, for example, is obtained by using different weights β'_x instead of β_x in (5.3): the derivatives of the B-spline base function at the subgrid positions, which are also stored in a lookup table. For a combined calculation of intensity and gradient, we reduce the 16 vectors once by applying the

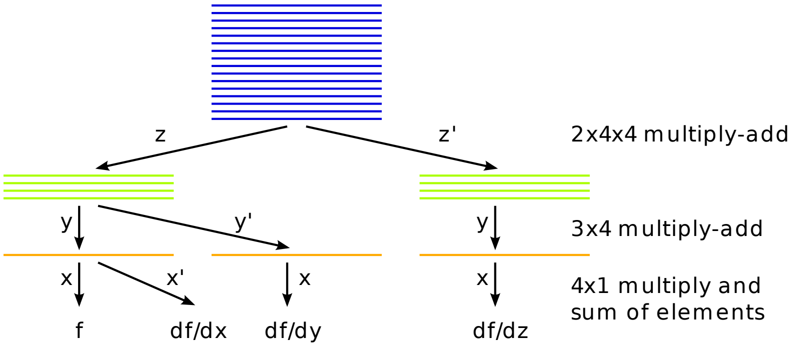


Fig. 5.3: Optimized image interpolation. Each colored line represents a SIMD vector containing four B-spline coefficients. For the concurrent evaluation of the image and its gradient at one coordinate, each of the 16 blue vectors is first multiplied with a vector containing the corresponding weights β_z. The results are added in groups of four to obtain the first set of four green vectors. Vectors containing coefficients at the same y coordinate are in the same group. A second set of four green vectors is obtained by repeating the previous steps with weights β'_z instead of β_z. Three different orange vectors are obtained by multiplying and adding the green vectors with weights β_y or β'_y respectively. The value f as well as its three derivatives can each be computed from the product of an orange vector with the corresponding β_y or β'_y by addition of its components.

weights β_z and another time using the weights β'_z in the first step, yielding two sets of the green vectors (Figure 5.3). The β_z-set is again reduced using both β_y and β'_y and the other set is reduced using β_y only. This yields three different versions of the orange vector based on different combinations of weights. To obtain the image intensity we use the vector based on β_z and β_y and do the 3rd step with weights β_x. The x-component of the gradient is calculated by applying β'_x to the same orange vector. Using weights β_x for the other two orange vectors yields the other two components of the gradient.

With this optimization, the required vector multiplications are almost halved from $4 \times (16 + 4 + 1) = 84$ to $2 \times 4 \times 4 + 3 \times 4 + 4 \times 1 = 48$ per point, and the vector load and assemble operations are reduced by a factor 4 if the moving image intensity and its gradient are calculated together.

5.2.6 Gradient Calculation

For the calculation of the derivative of the metric with respect to a coefficient we have to combine the contributions of all voxels in its support region. This was equation (4.26), which we repeat here:

$$\begin{aligned}\frac{\partial S(\mathbf{c})}{\partial c_i} =& \alpha \sum_{\mathbf{x} \in N_{c_i}} \sum_{\kappa=\kappa_0(\mathbf{x})}^{\kappa_0(\mathbf{x})+3} \log\left(\frac{p\left(\iota_0\left(\mathbf{x}\right), \kappa; \mathbf{c}\right)}{p_{mov}\left(\kappa; \mathbf{c}\right)}\right) \\ & \times \left.\frac{\beta_3\left(\tau\right)}{\partial \tau}\right|_{\tau=\kappa-\frac{f_{mov}(\mathcal{T}(\mathbf{x}|\mathbf{c}))-i_{mov,min}}{l_{mov}}} \\ & \times \left(\left.\frac{\mathrm{d} f_{mov}(\boldsymbol{\xi})}{\mathrm{d}\boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=\mathcal{T}(\mathbf{x};\mathbf{c})}\right)^{\mathrm{T}} \cdot \frac{\partial}{\partial c_i} \mathcal{T}\left(\mathbf{x}; \mathbf{c}\right), \end{aligned} \tag{5.4}$$

The corresponding part of the pseudo-code is in listing 5.2.

Listing 5.2: Pseudo-code loop A6

```
iterate over all transformation grid knots //
    loop A6
  iterate over all points in the neighborhood
    get fixed image intensity
    get moving image intensity and its gradient
    get probability density function derivative
```

```
        calculate the gradient of the metric at this
           point and add it to the total gradient at
           this grid knot
```

The outer loop is over all the knots of the transformation grid; the inner loop is over all the fixed image points in the neighborhood of that point. Each fixed image point is in the neighborhood of $4 \times 4 \times 4$ grid knots. For the respective computation of (4.26) at different grid knots, only the last factor

$$\frac{\partial}{\partial c_i} \mathcal{T}(\mathbf{x}; \mathbf{c}) \tag{5.5}$$

varies. This is the geometry-dependent part of the gradient. For the B-spline transformation model, it results in

$$\frac{\partial}{\partial c_{\hat{\imath},\hat{\jmath},\hat{k}}} \mathcal{T}(\mathbf{x}) = \frac{\partial}{\partial c_{\hat{\imath},\hat{\jmath},\hat{k}}} \sum_{i,j,k=0..3} c_{i,j,k} \beta_{x,i} \beta_{y,j} \beta_{z,k} = \beta_{x,\hat{\imath}} \beta_{y,\hat{\jmath}} \beta_{z,\hat{k}} \tag{5.6}$$

with the same weights β as for the computation of the transformation.

Reordering allows a more efficient processing. We move the loop over the image points outwards to get

Listing 5.3: Pseudo-code loop A6

```
iterate over all points of the fixed image
  iterate over all transformation grid knots in
     the neighborhood
    get fixed image intensity
    get moving image intensity and its gradient
    get probability density function derivative
    calculate the gradient of the metric at this
       point and add it to the total gradient at
       this grid knot
```

As a first consequence, we can reuse the computation of the geometry-independent part of the computations in (5.4) related to one $\mathbf{x}$

$$\begin{aligned}
&\sum_{\kappa=\kappa_0(\mathbf{x})}^{\kappa_0(\mathbf{x})+3} \log\left(\frac{p\left(\iota_0\left(\mathbf{x}\right),\kappa;\mathbf{c}\right)}{p_{mov}\left(\kappa;\mathbf{c}\right)}\right) \\
&\times \left.\frac{\beta_3\left(\tau\right)}{\partial\tau}\right|_{\tau=\kappa-\frac{f_{mov}(\mathcal{T}(\mathbf{x}|\mathbf{c}))-i_{mov,min}}{l_{mov}}} \\
&\times \left(\left.\frac{\mathrm{d}f_{mov}(\boldsymbol{\xi})}{\mathrm{d}\boldsymbol{\xi}}\right|_{\boldsymbol{\xi}=\mathcal{T}(\mathbf{x};\mathbf{c})}\right)^{\mathrm{T}}
\end{aligned} \tag{5.7}$$

to calculate the contribution of this $\mathbf{x}$ to all coefficients in its neighborhood. Furthermore, the fact that the loop over the voxels now is the outermost one allows a similar grouping like before. In contrast to the evaluation of the transformation function at one point, this operation is not a reduction of the 64 coefficients to a scalar value but the spreading of scalars (the geometry independent part (5.7)) to the 64 nonzero gradient entries in its neighborhood with different weights, meaning that the path in figure 5.2 is followed in opposite direction. This is reflected in the geometry-dependent part (5.6), where certain multiplications of B-spline weights are shared by multiple knots. First, we generate a vector for each point in the group. For instance, if we want to compute the gradient in x direction, we set all the elements of such a vector to the x-component of the image gradient at the corresponding point. This yields $16^3 = 4096$ vectors for a transformation grid spacing of 16 voxel units. After multiplying each vector with its corresponding weights β_x, sub-groups of 16 resulting vectors belonging to points with the same y and z coordinates can be added to obtain one vector, yielding a total of 16^2 vectors. After multiplying each of these vectors with weights $\beta_{y,0}$,$\beta_{y,1}$,$\beta_{y,2}$ and $\beta_{y,3}$ and collecting the sums for corresponding coordinates z, 4×16 vectors are obtained. The final step is multiplication and collection for weights β_z to obtain the 16 gradient vectors for this block. This again significantly reduces the multiply and multiply-add operations.

Moreover, the 16 gradient vectors have to be read and written back once for the entire group instead of once per fixed image point.

5.2.7 Conclusions

A large part of the algorithm runtime is spent with B-spline computations. In this section, some optimizations regarding these computations were introduced. These apply to the sequential nonrigid registration algorithm, but will also be beneficial for the parallelization. A detailed analysis of the impact on performance will follow in the next chapter.

5.3 Parallelization Platform

5.3.1 Profiling

We repeat the experiment from section 5.1.2 with the optimized sequential registration algorithm. This time we measure the time spent in each block of the pseudo-code in listing 5.1 (with restructured A6). The results in table 5.1 show that most of the processing time is distributed over loops A1 to A6. The gradient descent optimizer and the computation of the metric are only minor contributors to the total runtime. Although loops A1 and A4 do the same computation, A1 has a larger share of the total runtime. This is because loops A4 to A6 do not have to be executed for every iteration. If the preceding gradient descent step was unsuccessful, these loops are skipped because the gradient from the preceding iteration can be used.

task	% of iteration
loop A1	6.55
loop A2	37.30
loop A3	10.56
mi (lines 10 to 12)	0.006
loop A4	4.67
loop A5	26.00
loop A6	14.91
gd (line 23)	0.003

Tab. 5.1: The percentage of the time spent for each sub-task of the algorithm

5.3.2 Theoretical Limitations

Assuming that the problem size is fixed and P is the proportion of the algorithm which can be parallelized, Amdahl's law [7] implies that the maximum speedup using N processors is

$$S_N = \frac{1}{(1-P) + \frac{P}{N}}. \tag{5.8}$$

The upper bound to the speedup can be calculated as

$$S_\infty = \lim_{N \to \infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{(1-P)}. \tag{5.9}$$

The loops A1 to A6 account for 99.991% of the runtime. If they are all parallelized, the upper bound of the possible speedup is $S_\infty \approx 10000$. Because P usually depends on the problem size, this bound is only valid for the image size used for the measurements. But the numbers qualitatively show that good speedups can be achieved if all the loops are parallelized.

However, if just one loop cannot be parallelized, the predictions immediately become less optimistic. If, for example, A3 remains sequential, P becomes 0.8943 and $S_\infty \approx 10$.

5.3.3 Limitations for GPUs

Histogram computation (loop A3) is a task for which GPUs were known to be not suitable. The problem is that the calculation of the histogram requires per point-pair a data dependent memory read-modify-write operation of the form

$$\texttt{hist}[\texttt{fixed\_img\_intensity}][\texttt{floating\_img\_intensity}] += c, \tag{5.10}$$

where the value in one histogram bin is incremented by c. For our Parzen window approach, even four such operations are required per point pair. While this is a simple task for a CPU, it is difficult to map it to the GPU architecture.

As discussed in section 3.5, traditional GPUs read data from the device memory (vertices, textures), pass them through the graphics pipeline and write the result to the render target in device memory. In this scheme, the write accesses are deterministic: the result (the color of the pixel in the render target) depends on the computations in the graphics pipeline, but not the location to which it is written (target pixel coordinates). However, for the histogram computation, random (data-dependent) write access is required.

Another issue is that the device memory is external DRAM and thus has a relatively high latency. There are on-chip caches, but they are read-only. Therefore, the histogram needs to be accommodated in external memory. A further complication is that the histogram is at the same time input and output of the increment function.

An implementation by Green [33] required one pass over the entire image per histogram bin. All the values which are not in the respective bin of the pass are masked out. The number of values which are not masked out are accumulated and the result is written to the corresponding bin. For a histogram with 32^2 bins, a prohibitively high number of 1024 passes would be required. Khamene et al. [50] came to the conclusion that mutual information cannot be implemented on GPUs.

During the work on this dissertation, GPUs have evolved. The latest GPUs have an on-chip shared memory, which is writable and enables more efficient histogram implementations for small bin counts [82]. However, the relatively large joint histograms used by the algorithm do not entirely fit into the shared memory, which prohibits a single-pass solution. Moreover, as its name says, the shared memory is shared by a group of threads. Therefore, write accesses have to be synchronized, which imposes an additional overhead. However, with the trend toward more general purpose architectures, future GPUs may very well overcome these limitations.

5.3.4 Target Platform for Parallelization

GPUs and the Cell/B.E. processor have a high potential for number-crunching applications. They are also relatively cheap, compared to large multiprocessor systems for example. To exploit these architectures, algorithms need to be parallelizable and mappable to the memory hierarchy of these architectures. GPUs are even more specialized than the SPEs on the Cell/B.E. processor. In the preceding section, we have seen that the memory model of GPUs is not suitable for the computation of the mutual information. However, the profiling of the algorithm showed that loop A3, the computation of the mutual information, has a significant share of the total runtime and therefore the maximum speedup achievable by parallelization is not very high if loop A3 remains sequential.

Because of current limitations of GPUs, the Cell/B.E. architecture was chosen as the target platform for the parallel nonrigid registration algorithm. Furthermore, small clusters of this architecture shall be considered.

5.3.5 Master-Worker Programming Model

A suitable programming model to implement such algorithms on the Cell/B.E. architecture is having a master thread running on the PPE and a group of worker threads running on the SPEs. If a sequential implementation of the algorithm is available, it can directly be compiled for the PPE, which is a general purpose processor core. The computationally intensive parts of the algorithm are partitioned and offloaded to the worker threads. The master thread runs on the PPE and takes care of the remaining sequential part and the synchronization of the worker threads. The number of worker threads is equal to the number of SPEs. The size of an SPE's local store (LS) is limited and has to accommodate instructions and data. Worker threads therefore tend to be relatively simple compute kernels.

This programming model can also be applied to systems with GPUs, which usually consist of a general purpose processor and an attached GPU. The master thread would run on the general purpose processor and the worker threads on the GPU. If future generations of GPUs allow to efficiently compute the joint histogram, the same parallelization scheme could be used.

General purpose multicore architectures also support the master-worker programming model. Because the master thread usually does not work in parallel with the worker threads, on a chip with four cores a master thread and up to four worker threads would be instantiated with the operating system taking care of the thread scheduling.

5.3.6 Conclusions

The six loops A1 to A6 together account for 99.991% of the runtime of the iterative optimization process. This makes them candidate functions for parallelization. They should be offloaded to parallel worker threads. Target platforms for the implementation of the algorithm will be the Cell/B.E., which provides a very high computational performance compared to standard CPUs. GPUs are not considered for implementation because of known limitations for mutual information computation.

5.4 Data Locality

On the Cell/B.E. processor, data transfers between the SPE's local store and main memory are managed explicitly by the program and not by a hardware cache hierarchy. This means that complexity is removed from hardware and added to software. The advantage is that software is provided with the full flexibility to manage data locality and to control data-transfer granularity. Producer-consumer locality can also be exploited in order to avoid redundant write-backs of temporary results to main memory.

5.4.1 Producer-Consumer Locality

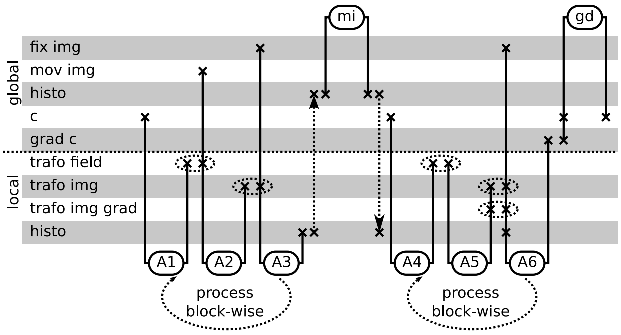


Fig. 5.4: Producer-consumer relationship for the different data structures. In loop A1, for example, the dense transformation field is computed based on the transformation coefficients. This field is then used in loop A2 to transform the moving image. Block-wise processing allows to keep the temporary result, the transformation field, in the local store in spite of its limited size.

In sections 5.2.4 and 5.2.6, we have seen that it is efficient to process groups of points of the fixed image together, if the computation of the transformation function for these points depends on the same B-spline coefficients. These coefficients are located on a regular grid. Therefore, the groups are cuboids of the fixed image, or cubes in the case of a uniform grid.

In the producer-consumer diagram (Figure 5.1), we see that in phase A1 the dense transformation field is calculated and written to main memory and then used in phase A2. Although these two loops will be offloaded to the SPEs, these temporary data have to be written to main memory because they are too large to be kept in the local store (or cache in the case of a general purpose processor). However, if we only process a subset of the image at a time, such that the temporary results fit into the local store, we can execute A2 after A1 without transferring the dense transformation field to main memory and back to local store. Memory transfers are further reduced by keeping the transformed moving image between phase A2 and A3 in the local store. Similar optimizations also apply between stages A4 and A5, as well as A5 and A6. For A6, this is possible because we restructured this block in section 5.2.6, such that the outermost loop iterates over all fixed image points. Figure 5.4 shows how the data structures are distributed between local stores of the SPE and global system memory. For now, we assume that loops A1 to A6 run on one single SPE and the phases “mi” and “gd” on the PPE. The phases are executed one after another. There is no thread-level parallelism.

We decide to process sub-blocks of $8 \times 8 \times 8$ voxels of the fixed image at a time. Furthermore, the transformation grid spacing is restricted to be a multiple of the image-block width or vice versa. Possible grid spacings are therefore 1, 2, 4, 8, 16, 24, 32, etc. voxel units. These restrictions allow to implement the optimized field and gradient computations within an image-block. However, the grid does not necessarily have to be isotropic. For example, a spacing of 16 voxel units in x- and y-direction and a spacing of 8 in z-direction is possible.

Choosing a power of two value as the block width allows to efficiently convert integer voxel coordinates to block coordinates. Instead of an integer division or multiplication, a shift operation can be used. For a block width of 8, the voxel coordinates have to be shifted 3 bits to the right to obtain the block coordinates.

To temporarily store the dense transformation field for a block as single precision float vectors, 8 kByte are required. Although a larger block size possibly allows better optimized computation (e.g. in the case of a grid spacing of 16), the local store would be too small. For a block width of 16, 64 kByte would be required to hold the temporary dense transformation field. Because the 256-kByte local-store also has to accommodate other data, for example image data and the instructions of the program, it is too small for a larger block size.

The entire histogram data structure is kept in the local store of the SPE during the block-wise processing of the loops A1 to A3. We restrict its size to a maximum of $64 \times 67 = 4288$ bins (67 bins because of the additional three bins for the padding for the moving image Parzen window). After processing the last block, the histogram is transferred to main memory to be used on the PPE.

5.4.2 Spatial Locality

During the processing of the (cubic) groups of fixed image points, we also have to fetch the fixed image intensities (e.g. for phase A3) and the moving image coefficients (e.g. for phase A2) for the entire group. This works especially efficient if the required data are located at neighboring memory locations. Hardware cache-mechanisms try to exploit the principle of locality of reference. If a particular memory location is accessed at a point in time, they expect neighboring memory locations to be accessed shortly afterwards. On the Cell/B.E., the SPEs load data by DMA transfers of variable length. However, a few large transfers are more efficient than many small ones [52]. Thus, on both platforms it is beneficial if a block of image data has a high spatial locality instead of being scattered in memory.

The fixed and the moving image are large 3D arrays of data. There are different ways how to store them in the one-dimensional memory address space. Row-major mapping means that the elements of the 2D array

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \end{pmatrix} \tag{5.11}$$

are stored like 0, 1, 2, . . . , 23 to linear memory. By block-2 mapping, we mean that the array is stored block-wise row-major, with a block width of 2. Thus, the same array would be mapped to memory in the following order: 0, 1, 6, 7, 2, 3, 8, 9, 4, 5, 10, 11, 12, 13, 18, 19, 14, 15, 20, 21, 16, 17, 22, 23.

In the context of 3D images, we assume that row-major means that if the image data are traversed linearly in memory, the x-coordinate changes fastest and the z-coordinate slowest. We choose to store fixed and moving image data in block-8 mapping because it improves spatial locality compared to row-major and also yields moderate block-sizes, such that multiple image

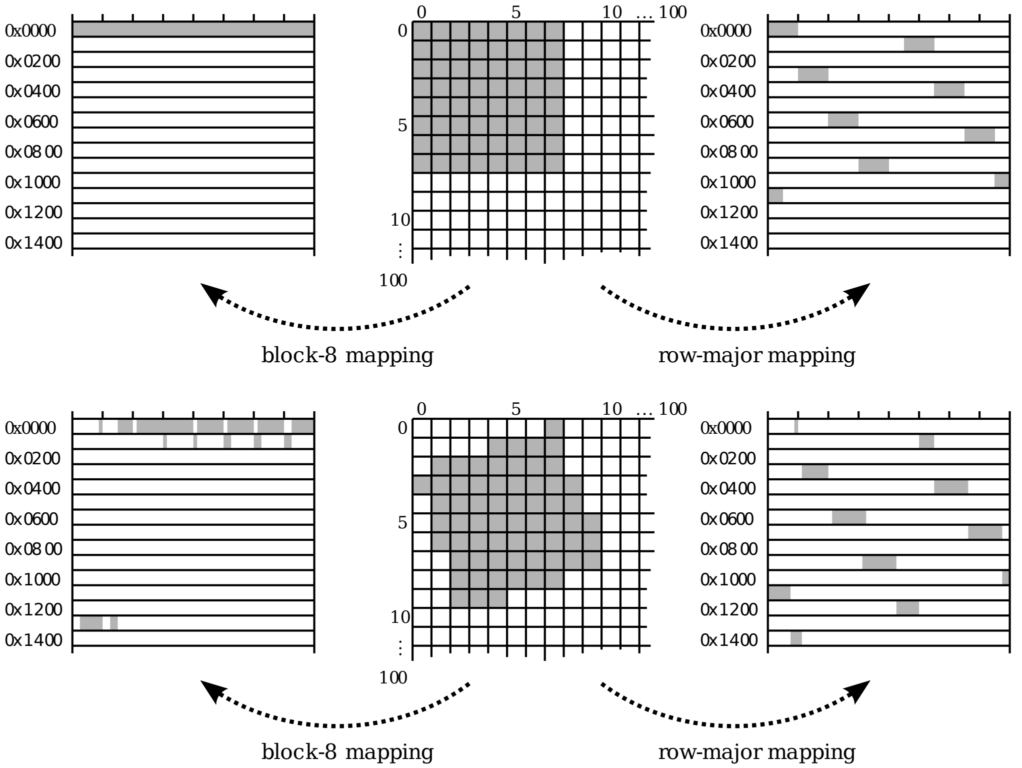


Fig. 5.5: This figure compares block-8 and row-major mapping for a 100^2 pixel 2D image. A fixed image block aligned like in the highlighted region in the upper part of the figure is required per pass of phase A3. While this leads to scattered memory accesses in the case of row-major mapping, all the required data are in a continuous memory block in the case of block-8 mapping. The moving image is accessed within the transformed fixed image region, which corresponds, for example, to the highlighted region in the lower part of the figure. Although the perfect alignment for these memory reads is lost, block-8 mapping still provides better locality than row-major mapping if we assume coarse-grained memory accesses of one line each.

blocks easily fit into the local store. If the image size in any dimension is not a multiple of 8, the image is padded so that all the blocks have the same size. Figure 5.5 shows the block-8 and row-major memory mapping of image blocks for the 2D case.

5.4.3 Storing Intermediate Results for Reuse

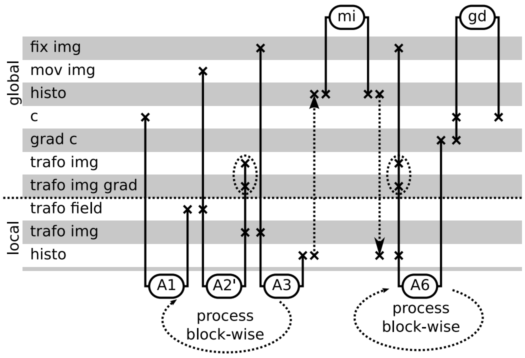


Fig. 5.6: Phases A4 and A5 can be removed if we compute the transformed moving image and its gradient in phase A2 and keep the results in main memory for use in phase A6.

There are redundant computations in the algorithm. In the producer-consumer diagram (Figure 5.4), phases A1 and A4 do identical computations. Furthermore, the transformed moving image is computed in phases A2 and A5. In phase A5, the gradient of the transformed moving image is computed additionally. As discussed in section 5.2.5, calculation of the transformed image and its gradient are done most efficiently together. We therefore do both computations already in phase A2 (now called A2') and temporarily store them in main memory for re-use in phase A6, which allows skipping of phases A4 and A5 (Figure 5.6). Phase A3 can still use the transformed moving image data directly from local store. We will see later that the transfer of temporary data to main memory can be hidden using double buffering and therefore does not impose any significant overhead. However, additional memory for the temporary data is required.

5.5 Parallelism 1: Multiple SPEs

5.5.1 Parallelization

After profiling, loops A1 to A6 were identified as the phases of the algorithm to be parallelized. For these parts, the data points in the fixed image are processed block-wise. This does not only improve the locality of data accesses, but it also allows to process one block of data independently from the others throughout A1 to A3 and then A4 to A6 (or just A6 if the temporary results are stored for re-use). For a relatively small image with 128^3 voxels, we get $16^3 = 4096$ blocks. If N_{WB} is the number of (work-)blocks and N_T the number of threads, $N_{WB} \gg N_T$ will be assumed. This allows to distribute the workload block-wise among the SPEs. Each block gets assigned to one SPE and the SPEs can work on their blocks in parallel.

In the next chapter, a performance analysis of the parallel registration algorithm will follow. This analysis bases on model functions for the runtime of each of the phases of the algorithm. In this analysis, the first group of parallelized loops (A1 to A3) are called "phase B1". A4 and A5 are not executed (temporary buffering is activated). For consistency, loop A6 is called "phase B2". The runtimes of these two phases are modeled using the functions

$$t_{B1}\left(N_P, N_I, N_T\right) = c_{B1}\frac{N_P N_I}{N_T} \tag{5.12}$$

$$t_{B2}\left(N_P, N_I, N_T\right) = c_{B2}\frac{N_P N_I}{N_T} \tag{5.13}$$

with constants c_{B1} and c_{B2} to be determined. N_P is the number of points in the fixed image; the runtime is expected to grow linearly with the image size. The runtime should also depend linearly on N_I, the number of iterations of the algorithm. A higher number of parallel threads N_T reduces the workload per thread. Thus, an inverse proportionality of the runtime can be assumed.

5.5.2 Compute Kernels

The functions that are offloaded to the SPEs are relatively simple and additionally they share multiple sub-functions related to B-spline computations. This allows us to implement one program capable of executing all these functions. At the start of the registration application, this program is loaded to each SPE, where it is continuously running. It receives messages from the master thread, for example “execute loop A6 on assigned work blocks”. After finishing, it sends a completion message back to the master thread, which waits until it received all completion messages. Having one single program running on the SPEs removes the overhead of loading a separate program each time a function is called. The assignment of work blocks to worker threads is static and defined at startup. All work blocks take roughly the same time for processing. Thus, not much can be gained by a dynamic load-balancing. There is less communication overhead for static assignment than, for example, in the case of a work queue. In such a setup, the master thread would add work blocks to a queue and the worker threads dynamically fetch the first element of the queue to process until the queue is empty. This requires communication between an SPE and the PPE for each work block.

5.5.3 Synchronization and Coherency

When multiple threads work on the same data structures in parallel, care has to be taken when data are modified, similar to shared-memory multiprocessor systems.

The fixed and moving image data are not modified throughout the registration, therefore no synchronization or coherency problems occur related to these two data structures. The transformation coefficients are only modified during “gd”. This phase runs exclusively on the PPE and therefore no synchronization is necessary. However, the SPEs cache these coefficients in the local store. To avoid coherency issues, cached coefficient data have to be invalidated at the start of every iteration.

Although the worker threads write to the temporary data (transformed moving image and its gradient), no synchronization is necessary because no two threads will write to the same location.

However, all the threads write to the histogram and the gradient of the metric. These are both relatively small data structures compared to, for example, the fixed image data. To avoid thread synchronization overhead, these arrays are replicated. Each thread has its private copy in main memory and writes the histogram of its subset of the image to its private histogram array. It also does so for the gradient of the metric.

5.5.4 Communication

This data replication imposes some additional overhead before phases "mi" and "gd". The master thread has to collect the partial results from the worker threads. In the case of the histogram, the master thread sends a message to each worker thread, instructing them to transfer their histogram array from the local store to main memory. When these transfers are finished, the master thread computes the sum of the partial results. The runtime of this function will be modeled as

$$\begin{aligned} & t_{hsync}\left(N_B, N_I, N_T\right) = \\ & \qquad \left(c_{hsync,1} + c_{hsync,2} N_T + \left(c_{hsync,3} + c_{hsync,4} N_T\right) N_B\right) N_I, \end{aligned} \tag{5.14}$$

with N_I being the number of iterations, N_B the number of histogram bins and N_T the number of worker threads. The parameters $c_{hsync,1}$, $c_{hsync,2}$, $c_{hsync,3}$ and $c_{hsync,4}$ are to be determined. $c_{hsync,1}$ and $c_{hsync,2}$ are related to the latency of messages and data transfers. $c_{hsync,3}$ and $c_{hsync,4}$ address the operations which do depend on the number of histogram bins, such as data transfers (additionally to their latency) and especially the computations.

Similar synchronization is necessary to collect the partial gradients and therefore we will model its runtime as

$$\begin{aligned} & t_{gsync}\left(N_K, N_I, N_T\right) = \\ & \qquad \left(c_{gsync,1} + c_{gsync,2} N_T + \left(c_{gsync,3} + c_{gsync,4} N_T\right) N_K\right) N_I, \end{aligned} \tag{5.15}$$

with N_K being the number of knots of the transformation grid.

5.5.5 Model Overview

In the preceding sections, we described the parallel stages of the algorithm and defined model functions for their runtime with parameters that we will have to determine later. Similar model functions were derived for the following communication phases. With the remaining sequential phases of the algorithm, the overall flow of one iteration consists of six phases 5.7.

To complete the introduced model, we define functions for the runtime of the sequential stages. The time required for the computation of the metric

$$t_{mi}\left(N_B, N_I\right) = c_{mi} N_B N_I \tag{5.16}$$

is proportional to the number of histogram bins and the number of iterations.

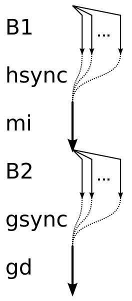


Fig. 5.7: One iteration of the parallel nonrigid registration algorithm has six stages. Two parallel computation stages (B1 and B2), each one followed by communication phases (hsync and gsync). The two sequential parts are "mi", computation of the mutual information metric and related calculations (lines 10 to 12 in listing 5.1), and "gd", the gradient descent optimizer.

The coefficients are updated per gradient entry and thus we estimate the runtime to be proportional to the number of knots N_K and the number of iterations N_I.

$$t_{gd}\left(N_K, N_I\right) = c_{gd} N_K N_I \tag{5.17}$$

5.6 Data Transfers

5.6.1 Double Buffering

Data to be processed by the SPE and results to be stored back are transferred between system memory and local store by DMA transfers. Data transfers are controlled by the MFC and can be in parallel to computations on the SPUs. Figure 5.8 illustrates how all data transfers can be in parallel to computations during phase B1.

Double buffering is also possible for phase B2 in a similar manner if no temporary data are stored. Otherwise we load all the data required to process a block (fixed image data and temporary data) during the computations of the preceding block.

Data accesses have to be predictable to enable prefetching. For most data, this predictability is given because of the static assignment of work blocks to worker threads. Each thread knows which block it processes next. What is not trivial to predict is the access pattern of the moving image data. The required moving image data are known only after completing the com-

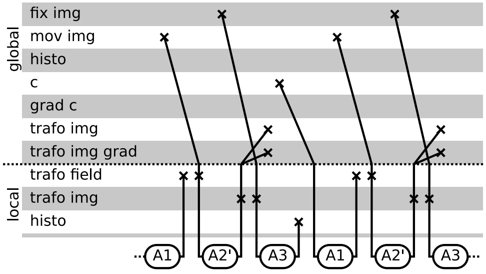


Fig. 5.8: Interleaved data transfers. During computations of A1, the moving image data required in A2' is transferred to local store. During phase A2', the fixed image data required in A3 is loaded. During A3, the temporary results, which were computed in A2', are written to main memory for later re-use. Also, the transformation coefficients required during phase A1 of the next block to be processes are prefetched.

putation of the mapped coordinates, but we would like these calculations and transfers to be done in parallel. Therefore, we will introduce a probabilistic approach to prefetching the moving image data in section 5.6.3. But first, we will discuss the layout of the data cache in the local store.

5.6.2 Caching of Data

The fixed image is processed in groups of 8^3 voxels, which is the granularity with which fixed and moving image data are stored. Thus, the image data caches in the local store are administered with this granularity as well.

The local-store cache layout during phase B1 has one slot for the fixed image, which contains the currently processed image block. If the voxel sizes of the fixed and the floating image are similar, we can expect a fixed image block to overlap with up to $3^3 = 27$ moving image blocks. The moving image cache is direct mapped[1] and has 27 slots. The moving image block with coordinates $\{B_x, B_y, B_z\}$ is mapped to the cache index

$$i = 9 \times (B_z \bmod 3) + 3 \times (B_y \bmod 3) + (B_x \bmod 3) \tag{5.18}$$

There is also one additional slot to store vector data (first the dense transformation field for one block, then the gradient of the transformed moving

---

[1] For a direct mapped cache, a cache line (in our case an image block) can be placed just in one specific slot of the cache. A fully associative cache, for example, would allow any cache line to be mapped anywhere in the cache

image of this block) and one slot to store scalar data (the transformed moving image).

In phase B2 (with storing of temporary data), no moving image cache is required. Part of the local store assigned to this cache can be used for the double buffering of fixed image and temproary data. To overlap computations and data transfers of two consecutive blocks, two slots for each fixed image, temporary scalar and temporary vector data are required.

The amount of transformation coefficient data required to process one block is much smaller than the amount of image data. Thus, we do not expect a serious performance impact if these data structures are managed at a much smaller granularity. For the 4^3 knots in a neighborhood, the values associated with adjacent knots in one direction can be placed to adjacent memory locations. Therefore, these data are moved by 16 DMA transfers of 64 Byte each (a 16 Byte vector of coefficients per knot).

5.6.3 Probabilistic Prefetching of Moving Image Data

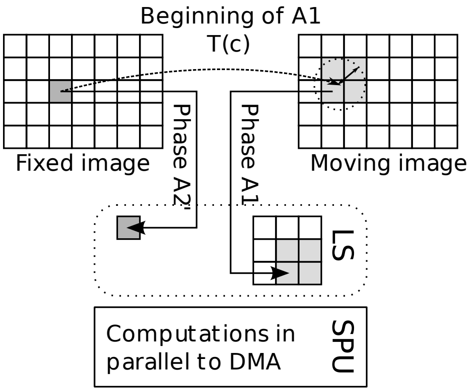


Fig. 5.9: During phase A1 (evaluation of the transformation field), the moving image blocks in a neighborhood of the mapped center of the currently processed fixed image block are prefetched. During phase A2' (interpolation of the moving image and its gradient), the fixed image data are prefetched. They will be used in phase A3 to update the joint histogram.

The moving image data are stored in a cache with 27 slots. It is the only data structure, to which accesses are not exactly predictable. In a typical cache, this would mean that as soon as an access to moving image data not available in the cache is detected, the block containing this data is fetched.

This causes a cache-miss penalty because the program has to wait until the data are available.

To detect if the data are available, the program has to first find out in which slot they would be stored (using 5.18). Each cache slot also has a tag field, which contains the coordinates of the block currently stored in this slot. Thus, the program has to compare this tag to the block coordinates it tries to access. If they are not equal, the data are fetched and the tag is updated.

To reduce the number of cache misses, moving image blocks which will be used in A2' with a high probability are prefetched already during A1. To decide which blocks of the moving image to fetch, we first calculate the mapping of the point in the center of the fixed image block, $\mathbf{c}' = \mathcal{T}(\mathbf{c})$ (Figure 5.9). The block containing $\mathbf{c}'$ and its 26 neighbors are prefetched. This should include all the required moving image data to process the current fixed image block, assuming that the voxel size in both images is the same and that the transformation $\mathcal{T}$ does not distort too extremely into any particular direction. If we process neighboring fixed image blocks one after another, the probability that a large part of the required moving image blocks for the current fixed image block were already fetched when processing the preceding fixed image block is high and we do not have to re-fetch them. However, with this approach it is also possible that we fetch blocks that are not needed and thus we unnecessarily stress the memory bandwidth. To ensure that our implementation does not become memory bound due to excessive prefetching, we introduce the threshold parameter d_{thr} and only fetch blocks with a center point $\hat{\mathbf{c}}$ meeting the condition $\|\hat{\mathbf{c}} - \mathbf{c}'\| < d_{thr}$. A lower threshold results in fewer moving image blocks being prefetched. If the threshold is set too low, cache misses can occur in phase A3. In this case, the SPU stalls while waiting for the missing moving image data to be transferred into the local store.

5.6.4 Data Replication

To interpolate the moving image at arbitrary coordinates, the image coefficients within a 4^3 neighborhood are required. Because the moving image data are stored block-wise, this data may be distributed over up to eight moving image blocks. The performance analysis in the next chapter shows that the loading of these moving image coefficients to the vector register-file of the SPEs can impose a significant overhead if not implemented efficiently.

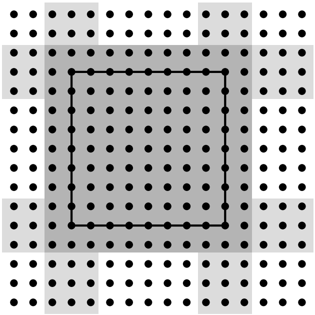


Fig. 5.10: The moving image data (the coefficients represented as circles) are stored in overlapping blocks. To evaluate the moving image at arbitrary coordinates within the solid rectangle, the coefficients within the dark-gray rectangle are possibly required and thus stored in one block. The light-gray stripes are the overlap-regions, which can be found in multiple blocks.

One condition for an efficient implementation is that we can easily locate the required coefficients in memory. If the neighborhood crosses block boundaries, computation of the memory addresses of the coefficients becomes more complicated. This problem can be avoided by replicating moving image coefficients (Figure 5.10). Doing this, all the coefficients required to interpolate the moving image at one coordinate can be found within one moving image block. Thus, the block size increases from $8 \times 8 \times 8$ to $11 \times 11 \times 11$, or by a factor of almost 2.6. If the implementation is memory bound, this trade hardly would make sense. However, the performance analysis will show that a significant speedup can be achieved by optimizing the load function. Furthermore, it will show that memory bandwidth is no issue on current architectures.

5.7 Parallelism 2: Multiple Cell/B.E. Processors

5.7.1 The Parallelized Phases

On a small cluster of QS21 Cell/B.E. blades (the nodes), each featuring two processors, the parallelization scheme is similar to the one discussed in the preceding sections. On each node, there is one master thread controlling its up to 16 worker threads. If we have four blades, this results in a total of up to 64 worker threads sharing the workload of phases B1 and B2. The assignment of work blocks to worker threads is still static.

5.7.2 The Communication Phases

The reduced workload per thread should result in a speedup of phases B1 and B2. However, additional communication is required to keep the blades in sync. The fixed and moving image data structures are not modified throughout the registration and we load them to main memory of each node during the startup of the algorithm.

After each master thread has collected the partial histograms of its worker threads, the master threads exchange these intermediate results such that each one obtains the full histogram and can continue with the computation of the mutual information. MPI is a popular API for communication on clusters, where the nodes cannot access a shared memory space but have to exchange data by message passing. The `MPI_Allreduce` function, which is defined by the MPI API, implements the communication pattern which we need for this synchronization. It collects data from all the nodes, performs an operation on them and distributes the results back to all nodes. Various operations would be supported, but in this case an addition is required. After collecting the partial gradients of the metric from their worker threads, the master threads also have to synchronize these arrays with an `MPI_Allreduce` operation.

Because the assignment of work blocks is static, the temporarily stored values are re-used by the same worker thread that produced them. Thus, no additional communication between the nodes is required because this data do not have to be exchanged. This is possible because in section 5.2.6 we restructured phase A6, such that the outer loop goes over all fixed image points. Without this reorganization, the outermost loop would go over all transformation grid knots. For parallelization, each knot would be assigned to a worker thread and thus to a node. Neighboring knots depend on overlapping image regions for their computations. Knots assigned to different nodes would require the same transformed moving-image data (and their gradient). Either computations in these overlap regions would have to be carried out on multiple nodes or the nodes would have to exchange their results. Because these data structures are much larger than the gradient of the metric, the chosen workload distribution is more efficient.

The Allreduce Operation

`MPI_Allreduce` is a combination of a reduce operation, where the result is available on one master node, and a broadcast operation to distribute this result. Such all-to-all communication is one of the most common communication patterns in high performance computing. There are different `MPI_Allreduce` methods and the right choice of the algorithm and its parameters can have a significant impact on performance. The decision does not only depend on the application, but also on the system. Important system parameters include the ratio between the network and the processor speeds, the switch design, the amount of buffer memory in switches, and the network topology [29]. A discussion of these topics goes beyond the scope of this dissertation. We will use *OpenMPI* [3] to handle the inter-node communication.

Modeling the Cost of the Allreduce Operation

Hockney's model for *point-to-point* communication performance depending on the message length m

$$t(m) = t_0 + \frac{m}{r_\infty} \tag{5.19}$$

was extended for collective communication in multiprocessor systems by Xu et al. [108] by making the startup time t_0 and the asymptotic bandwidth r_∞ dependent on the number of compute nodes n.

$$t(n, m) = t_0\,(n) + \frac{m}{r_\infty\,(n)}. \tag{5.20}$$

t is the time required for the communication operation. The asymptotic bandwidth is the maximal bandwidth achievable when the message length approaches infinity. While t_0 is dominating for small message sizes, the performance is mostly depending on r_∞ for large messages. In experiments on an IBM SP2 system, they modeled

$$t_0\,(n) = c \log_2 n + d, \tag{5.21}$$

where c is the additional time required for the reduce operation with very small messages when the number of nodes is doubled. The asymptotic bandwidth for the reduction was not measured. Formulae were derived by Gunawan et al. for Myrinet based systems [36] and by Chou et al. [16] for Gigabit Ethernet based clusters. Chou et al. also observed a logarithmic behavior for the reduce operation. They model the total communication time as

$$t\,(n,m) = c\log_2 n + d + (e\log_2 n + f)\,m \tag{5.22}$$

Building the interconnect based on Ethernet is attractive because of its availability and low cost, but traditionally has a major drawback compared with typical cluster interconnects like InfiniBand or Myrinet, which is its much higher latency. This is because of the overhead of the TCP/IP software stack, which is used for MPI over Ethernet. For InfiniBand, the corresponding communication protocol layers are implemented in hardware.

Changes to the Model

In addition to the intra-node synchronization of the partial results, we have inter-node communication, which is modeled like

$$\begin{aligned} t_{\text{Allreduce}}\,(n,m) = {} & (c\log_2 n + d) + \\ & (e\log_2 n + f)\,m \end{aligned} \tag{5.23}$$

with n being the number of nodes, m the size of the data structure and c, d, e, f system parameters to be determined. The functions for t_{hsync} (5.14) and t_{gsync} (5.15) need to be adapted. If $N_{T,l}$ is the number of local threads per node and N_N is the number of nodes ($N_T = N_{T,l} \times N_N$), we use the two model functions

$$t_{hsync,cluster} = t_{hsync}\,(N_B, N_I, N_{T,l}) + t_{\text{Allreduce}}\,(N_N, N_B \times 4\text{Byte}) \tag{5.24}$$

$$t_{gsync,cluster} = t_{gsync}\,(N_K, N_I, N_{T,l}) + t_{\text{Allreduce}}\,(N_N, N_K \times 16\text{Byte}) \tag{5.25}$$

for our Cell/B.E. cluster.

5.7.3 The Sequential Phases

When entering the two sequential phases of the algorithm, all the data structures on the different nodes are in sync. Each master thread can calculate the metric or do the gradient descent step independently of the other nodes.

5.7.4 Conclusions

The Cell/B.E. cluster offers 64 processing units by only connecting four QS21 nodes. This allows to speed up the computational stages of the algorithm, but the synchronization overhead will also increase. In contrast to the (shared-memory) multicore parallelization, communication over the network is required. This communication will be implemented with MPI functions. Because the algorithm was restructured such that all parallelized stages are loops over the fixed image points, rather than the transformation coefficients, only the relatively small histogram and the gradient of the metric need to be synchronized. No transformed moving image data need to be exchanged by the nodes.

6

Performance Analysis

This chapter is dedicated to an analysis of the performance and scalability of the parallel nonrigid registration algorithm. This algorithm was designed for the Cell/B.E. processor, which was mainly competing with single-core x86 processors when it first came to the market in 2005. Although technology has advanced in the meantime and possible transistor densities have increased considerably, no Cell/B.E. architecture with increased number of cores has been released. However, in the meantime x86 processors also have started to evolve into multicore processors with increasing number of cores. The chosen programming model allows to port the algorithm with little effort to x86 multicore processors. The main difference is that the explicit DMA commands need to be removed and the data can be accessed transparently in main memory.

In the first section, the performance of the algorithm on the Cell/B.E. processor and on two different x86 architectures is compared. The first x86 processor is from the same generation as the Cell/B.E., while the second one is a more recent processor. Furthermore, the impact of the different optimizations discussed in the preceding chapter is analyzed. In the second section a detailed analysis of the scalability on the Cell/B.E. processor follows. It bases on the model functions for the runtimes of each phase of the algorithm, which were defined in the preceding chapter. The following estimation of the performance on future processor generations beyond eight cores is based on the performance models derived for the scalability analysis. The goal is to identify possible performance bottlenecks on many-core processors. The communication model and the memory bandwidth requirements come under scrutiny. Finally, scalability on systems consisting of multiple Cell/B.E. processors is analyzed. A Gigabit Ethernet cluster of four QS21 blades, each containing two Cell/B.E. chips, is used for these experiments.

6.1 Benchmark

6.1.1 Setup

The implemented method will be called the streaming algorithm because it bases on the streaming programming model of the Cell/B.E. processor. To compare different processors and the effect of the optimizations discussed in the preceding section, the runtime of the streaming algorithm for 22 image pairs was measured. The smallest fixed image had a size of $256 \times 256 \times 30$ voxels, the largest one $512 \times 512 \times 124$ voxels. The runtime was divided by the number of voxels and the number of iterations (30) and the mean of this value over all the 22 experiments was calculated. Multiresolution was disabled.

To evaluate the Cell/B.E. architectural characteristics, its performance is first compared to an Intel Xeon processor of the same generation (90 nm technology). To see how x86 processors have evolved in the meantime and to see if the Cell/B.E. processor is still competitive, a more recent Intel Core2 Extreme quad-core processor is also included in the benchmark. We use a single-chip system in case of the Core2 processor. For the other two architectures, a dual-processor system with a shared memory space is used. For our communication model, it does not matter if two threads are running on two different cores of the same chip or on two different chips. Therefore, the algorithm also runs on such shared memory multiprocessor systems.

We also compare the streaming registration to an algorithm based on `ITK` (Insight Toolkit) [2]. `ITK` is an open source toolkit for image registration and segmentation. It provides building blocks that can be combined to form a complete registration algorithm. The `ITK` algorithm used in these experiments bases on the components corresponding to the features of the streaming algorithm: mutual information (using all sample points to calcu-

Description	Xeon 3.6 GHz	Core2 3.2 GHz	Cell/B.E.
ITK	62678.5	28698.1	-
Restructuring	3326.3	1067.6	-
Temp buffering	2552.6	813.6	-
Field grouping	2075.5	642.5	-
Gradient grouping	1259.8	407.9	-
Image LUT	1083.4	351.0	1806.3
Manual vectorization	789.9	243.5	669.5
Optimized load	-	-	333.8
Parallelism 1 Chip	-	63.3	43.5
Parallelism 2 Chips	401.4	-	23.14
Double Buffering	-	-	23.11

Tab. 6.1: Runtime per voxel and iteration in nanoseconds depending on optimizations and platform. The Intel Xeon processor was manufactured in 90 nm technology like the Cell/B.E. processor. The Core2 is a more recent x86 processor of the 45 nm generation.

late the histogram), a cubic B-spline transformation model, cubic B-spline image interpolation and a gradient descent optimizer.

6.1.2 Results Overview

Table 6.1 shows an overview of the algorithm runtime per voxel and iteration on the three systems for different optimization levels. The largest part of the speedup — almost 80× — is attained by efficient implementation (Figure 6.1). The effects of parallelism and differences in processor architecture are smaller. For one specific image of size 512 × 512 × 98 and including all the optimizations, the registration on 1 Cell/B.E. processor takes 34.5 seconds. Using the two processors, the runtime is reduced to 18.1 seconds. On the Xeon machine, the same registration problem takes 8 minutes and 20 seconds when using one CPU, and 4 minutes and 14 seconds when using both. On the Core2 using four threads it finishes after 42.8 seconds. The runtime below a minute on the Core2 and the Cell/B.E. processor stands in contrast to the 6 hours and 8 minutes required to register the same image pair with the `ITK` algorithm on the Core2 machine. In the following sections, we analyze the impact of the different optimizations on performance.

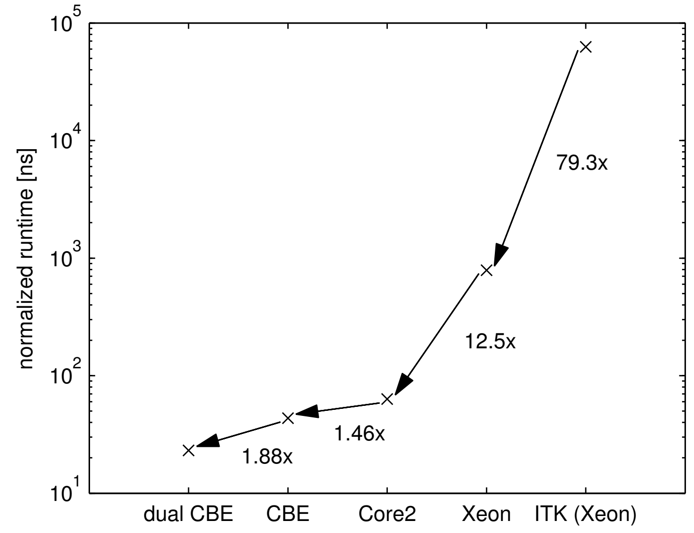


Fig. 6.1: The normalized runtime for different setups. More efficient programming yields a speedup of almost 80×. This is a large improvement compared to the factor 12.5 between the Xeon and the quad-core Core2. The difference between Cell/B.E. processor (CBE) and Core2 is relatively small.

6.1.3 Code and Data Structures

The code and data structure reorganizations that are necessary for the implementation on the Cell/B.E. architecture can also be beneficial for the x86 implementation (Table 6.1, first two rows). A first factor is the overhead of the `ITK` algorithm coming from its modular and flexible design. A second factor is the increased locality of the data structures. Removing `ITK` overhead and redundant computations accounts for a large part of the total speedup of almost 27× on the Core2 processor, but a non-negligible factor also comes from the more efficient data access patterns. While the `ITK` implementations stalls during 17.21% of the cycles[1], this fraction is only 2.86% for the restructured algorithm. In other words, just the stall time of the `ITK` implementation is more than 4.5 times higher than the total runtime of the streaming implementation. High stall ratios are often a result of a large number of cache misses. A cache miss is a failed attempt to read (or write) data in the cache, resulting in an access to main memory (or a higher level

---

[1] The number of cycles in which no micro operations were dispatched from the reservation stations was measured.

cache) with a higher latency. Execution of instructions depending on this data is delayed. For the `ITK` implementation, we could observe 119× more L1 cache misses[2] than for the streaming algorithm.

More efficient hardware prefetching is one possible explanation for the lower cache miss rate. The Intel Core2 processor implements multiple hardware prefetch mechanisms [28]. One of these mechanisms, for example, observes the distance in address space of memory accesses. It detects sequences of accesses that follow a regular stride and prefetches data along this stride, hiding the latency to main memory. Intuitively, this may be very efficient when traversing multi-dimensional regular grids, like 3D images, where neighbors along all but one coordinate axis are not stored in adjacent memory regions but are separated by a constant distance. However, the prefetch unit has difficulties to detect very large strides. One possible explanation for the lower cache miss ratio of the streaming algorithm is a more efficient prefetching because of the blocked image data structures, where neighbors within a same block are also closer in memory and therefore the strides are smaller when traversing a block. But when the hardware prefetch mechanisms are disabled, the total runtime only increases by 0.5% (`ITK`) and 1.5% respectively and also the number of cache misses only rises by a few percent.

For both algorithms, the cache miss rate and also the runtime do not change significantly if hardware prefetching is disabled. Therefore, the lower cache miss ratio can be attributed to the better locality of the data access patterns.

6.1.4 Re-use of the Transformed Image and its Gradient

If the size of main memory permits, storing the transformed floating image and its gradient during histogram calculation brings a significant improvement. When disabling this feature on the Cell/B.E. processor, the runtime per voxel and iteration increases from 23.11 ns to 34.12 ns. The runtime of the first parallelized phase, t_{B1}, decreases from 18.53 ns to 16.96 ns, because the gradient of the transformed moving image is not calculated any more. The small difference in runtime shows that the combined calculation of the image and its gradient is efficient (see section 5.2.5). Because the temporary

---

[2] The number of retired load operations that missed the L1 data cache was measured.

data is not stored and therefore needs to be computed again, the runtime of the second parallelized phase, t_{B2}, increases from 3.76 ns to 16.62 ns.

6.1.5 Grouping

The group-wise computation of the dense transformation field (see section 5.2.4) and the gradient (see section 5.2.6) together account for a speedup of about 2×. The group size was $8 \times 8 \times 8$ voxels, which is the size of a work block and thus cannot be further increased. However, the transformation grid spacing of $16 \times 16 \times 16$ voxel units would allow to process larger groups if more local store would be available. Although the block size cannot be increased on the Cell/B.E. processor, the impact of the block size on performance can be analyzed on the x86 platform.

Image Block Size

When increasing the block size, two opposing effects can be expected. On the one hand, larger groups of points can be processed under the condition that the transformation grid is at least as coarse as the image block size, which possibly reduces the runtime. On the other hand, the positive effect of increased locality may be lost.

In an experiment, we measure the time required to transform the moving image of size 256^3 depending on the block size for different transformations. The transformation is rigid: it is a rotation around an axis parallel to the y-axis through the center of the image and all the B-spline coefficients are set to zero. The B-spline grid spacing is equal to the image block width. Thus, a larger block size should result in more efficient grouping. In figure 6.2, we see that the runtime indeed decreases for larger block sizes. A block width of four yields roughly 1.4× larger runtimes than a block width of eight. This effect also applies for block widths of 16 and 32. However, for larger block sizes the runtime starts to rise again due to the decreased locality. While the runtime is almost independent of the rotation angle for small block sizes, large rotations yield significantly slower transformations for a block size width of 128.

In another experiment, nonrigid transformations with a grid spacing of 16 are applied to an image. Random Gaussian blobs are used to initialize the B-spline coefficients. The runtime of the transformation of the image for different amplitudes of the blobs is measured. In figure 6.3, we can

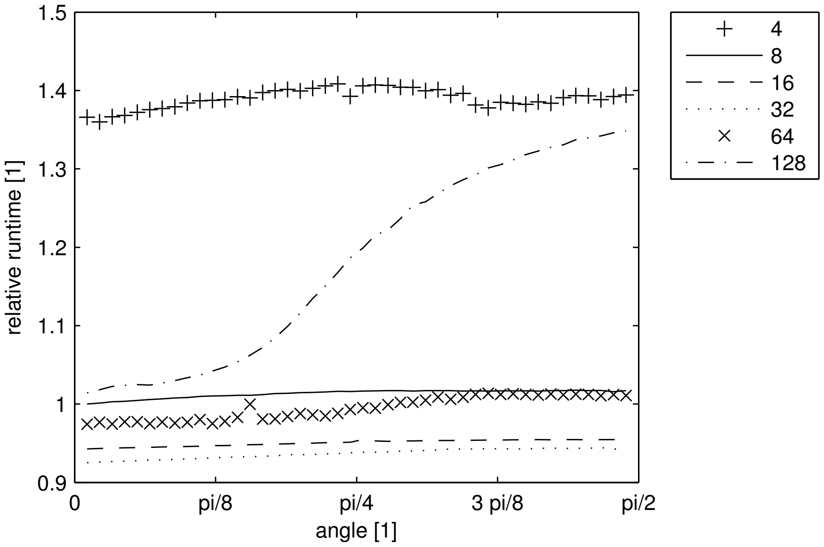


Fig. 6.2: The time required to calculate the transformed moving image depending on the transformation function. The runtimes are normalized using the fastest runtime for a block width of eight voxels. In this experiment, a transformation grid spacing equal to the image block size was chosen and all the B-spline coefficients were set to zero. There was only a rotation around an axis parallel to the y-axis through the center point of the image. The runtime does not depend on the rotation angle for small block sizes. For a block width of 128 voxel units, the performance decreases as the rotation angle grows.

observe similar effects as in the preceding experiment. First, the runtime decreases with increasing block size due to more efficient grouping. For too large block sizes, it increases again because the locality benefit is lost.

For these experiments, the Intel Core2 Extreme processor with four worker threads was used. For this setup, a block width of 32 yields slightly improved performance compared to a block width of eight voxel units. However, what was not considered in this experiment is that for larger block sizes, a balanced load distribution to the worker threads may become more complicated. Therefore, the relatively small performance benefit may not justify an increased block size, and a block width of 8 seems reasonable.

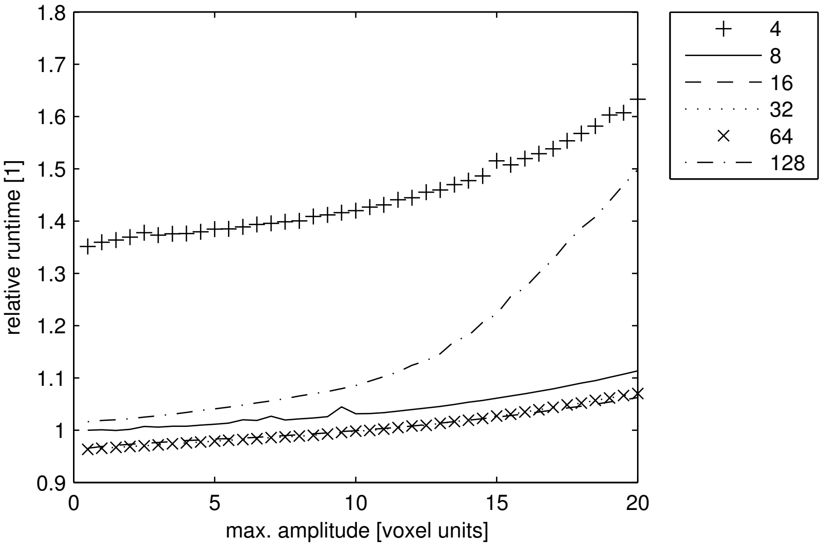


Fig. 6.3: The time required to calculate the transformed moving image depending on the transformation function. The runtimes are normalized using the fastest runtime for a block width of eight voxels. In this experiment, a transformation grid spacing of 16 voxel was chosen and the parameters were set using Gaussian blobs with varying maximum amplitude (x-axis). Larger transformations can result in longer runtimes. In the experiment with a block size of 128, the runtime is especially sensitive to the amplitude of the transformation.

6.1.6 Manual Vectorization

The `GNU Compiler Collection` (GCC) [1] was used to compile the algorithm for both x86 and the Cell/B.E. processors. Although this compiler tries to automatically make use of the vector processing capabilities of both platforms, significant improvements were achieved by manual fine-tuning of the vector operations. For example, the calculation of the sum of two vectors with four elements is written as a loop

```
int i;
float a[4], b[4], c[4];
...
for (i = 0; i < 4; i++) c[i] = a[i] + b[i];
```

in sequential C code. Instead, we can use vector data structures and intrinsics on the SPU:

```
vector float a, b, c;
...
c = spu_add(b, c);
```

and on the x86:

```
__m128 a, b, c;
...
c = _mm_add_ps(b, c);
```

More complex intrinsics exist, for example to rearrange the elements of a vector. On the Cell/B.E., the performance benefit was 2.70×. The SPU was designed to be especially efficient for vectorized code; it is not optimized to work on scalar or non-aligned data. On the x86, manual vectorization still resulted in a speedup of 1.44×.

6.1.7 Vector Alignment

On the SPEs, vector data can only be loaded from the local store to the register file if its address in the local store is 16-Byte aligned. The same condition applies to storing vector data from the register file to the local store. Otherwise the vectors have to be assembled explicitly, for example with the `spu_insert` intrinsic, which allows to set a particular element of a vector. Unaligned loads, however, can be much more expensive than aligned ones. Without data replication, the transformation coefficients and the gradient cannot be stored such that only aligned accesses are necessary, but the performance impact of the unaligned accesses is relatively small because the transformation coefficients and gradient entries only have to be transferred once between local store and register file per group and not per point (sections 5.2.4 and 5.2.6). However, the floating image coefficients have to be loaded per-point and alignment cannot be guaranteed either.

An improved function to load a vector from an unaligned address `src_addr` into the register file uses two aligned loads and assembles the result vector from these two vectors using a shuffle operation (`si_shufb`)

```
qword offset = si_andi(src_addr, 0x0f);
qword src_lo = si_lqd(src_addr, 0x00);
qword src_hi = si_lqd(src_addr, 0x10);
qword shift_by_bytes = si_shufb(offset, offset, vec_0x03);
qword shuffle_pattern = si_a(shift_by_bytes, vector_bytenr);
qword result = si_shufb(src_lo, src_hi, shuffle_pattern);
```

where `vector_bytenr` is a `char` vector with each element set to its index (`0x00, 0x01, 0x02, ...`) and `vec_0x03` is a `char` vector with each element set to `0x03`. By loading the floating image coefficients with this function, an overall improvement from 44.07 ns to 23.14 ns per voxel and iteration was achieved.

6.1.8 Double Buffering

Double buffering, which can be an efficient technique to improve performance by hiding memory latency, has virtually no impact on the performance of the algorithm. There are two possible reasons for this surprising behavior: either the double buffering does not successfully hide memory latency because the computations that are in parallel to the memory transfers take much less time than the transfers or the memory latencies are not an important factor for the performance. In section 6.4.4, we will see that the latter is the case.

6.1.9 Parallelism

Two important performance metrics in parallel computing are the speedup

$$S_p = \frac{T_1}{T_p} \tag{6.1}$$

and the efficiency

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}, \tag{6.2}$$

where p is the number of threads, T_1 the runtime for one thread and T_p the runtime for p threads. Algorithms with linear scalability have an efficiency of 1 (and a speedup of p). For the Cell/B.E., we get $S_8 = 7.68$ and $E_8 = 0.96$. For the Core2 processor, we get $S_4 = 3.84$ and $E_4 = 0.96$ and for the dual Xeon system $E_2 = 0.98$. This confirms that the chosen parallelization strategy is efficient for the Cell/B.E. as well as x86 processors. A detailed analysis of the scalability follows.

6.1.10 Conclusions

While the Cell/B.E. processor outperforms the same-generation x86 by more than an order of magnitude, more recent general purpose processor generations have almost closed this gap again. However, the basic streaming

Parameter	**Description**
N_P	Number of point pairs
N_I	Number of iterations
N_K	Number of transformation grid knots
N_B	Number of histogram bins
N_T	Number of worker threads

Tab. 6.2: Parameters of the runtime model

algorithm outperforms an `ITK` implementation by almost 27× (comparing the first two rows in table 6.1). Including the optimizations applying to the x86 platform, the speedup is 118× even without parallelism. Although many of the optimizations were inspired by properties of the specialized Cell/B.E. hardware, they are equally beneficial for x86 processors.

Compared to this, the speedup attributed to parallelism, which comes close to the number of processor cores, is relatively low. Therefore, an in-depth understanding of the algorithm, allowing an efficient (sequential) implementation, remains very important. The Cell/B.E. architecture used to have an advantage because of its relatively high number of cores. As the number of cores on standard CPUs grows, this advantage vanishes. Parallelism undoubtedly becomes an increasingly important factor. In the following sections, the scalability of the streaming algorithm will be analyzed in order to verify that in the future it can benefit from processors providing more parallelism.

6.2 Modeling the Runtime

In the preceding section we have seen that the algorithm efficiently uses the eight SPEs available on one Cell/B.E. processor and the experiments using the dual processor system show that it continues to scale well. Considering the trends in processor architecture development, we can expect processors with more cores to be available in the future. By attempting to extrapolate performance on larger systems based on measurements on similar, smaller systems, one can draw very misleading conclusions if no scalability analysis is performed [54]. In order to be able to estimate the behavior of the algorithm on future multicore processors, we break it down into different stages and express the runtime of each stage as a function of the parameters in table 6.2.

Stage	**Description**
t_{B1}	Phase B1 as defined in section 5.5.1
t_{hsync}	Collecting the partial histograms from the worker threads
t_{mi}	Histogram normalization and calculation of the metric
t_{B2}	Phase B2 as defined in section 5.5.1
t_{gsync}	Collecting the partial gradients from the worker threads
t_{gd}	Updating the coefficients using the gradient descent optimizer
t_{rest}	All the other processing

Tab. 6.3: The stages of the algorithm. Phases B1 and B2 are the computationally intensive parts that were offloaded to the worker threads. The synchronization of the threads takes place in t_{hsync} and t_{gsync}. t_{mi}, t_{gd} and t_{rest} are the parts of the algorithm that were not parallelized.

In the preceding chapter, we defined model functions for the runtime of each stage of the algorithm (see also table 6.3). Based on measurements with 22 image pairs of different sizes, the parameters of these model functions are estimated. N_P was between 2'097'152 and 33'554'432 point pairs. The basic settings were $N_I = 30$ and $N_B = 32 \times 35 = 1225$ (32 bins for the range of fixed image intensities, 32 plus 3 additional padding bins due to the Parzen window width for moving image intensities). The size of the transformation grid (N_K) was chosen depending on the image size with a default B-spline knot spacing of 16 voxel units. Each image pair was registered 20 times and the geometric mean of the 20 measured runtime values was used.

6.2.1 Measuring Runtime

To measure the time spent in each stage, the time when the master thread enters and leaves a stage was recorded and the difference was calculated. Because each stage usually is passed multiple times, the total time spent in each stage is accumulated. For these measurements, the `MPI_Wtime()`[3] function with a granularity of $1\mu s$ is sufficiently accurate. For some of the more detailed experiments in the later sections, a better time resolution is required. In these cases, the PPE time base register with its granularity of 37.5 ns is used. It can be read using the `__mftb()` intrinsic. For measurements on the SPEs, the decrementer register is used. The value in

---

[3] The `MPI_Wtime` function returns the elapsed time on the calling processors. It is defined by the MPI API

this register decrements at 26.667 MHz on the QS21 and therefore again a granularity of 37.5 ns can be achieved. It is accessed with the read and write channel intrinsics: `spu_writech(SPU_WrDec, 0x7fffffff)` sets the decrementer to the value `0x7fffffff` and `spu_readch(SPU_RdDec)` reads its content.

6.2.2 Parallelized Functions

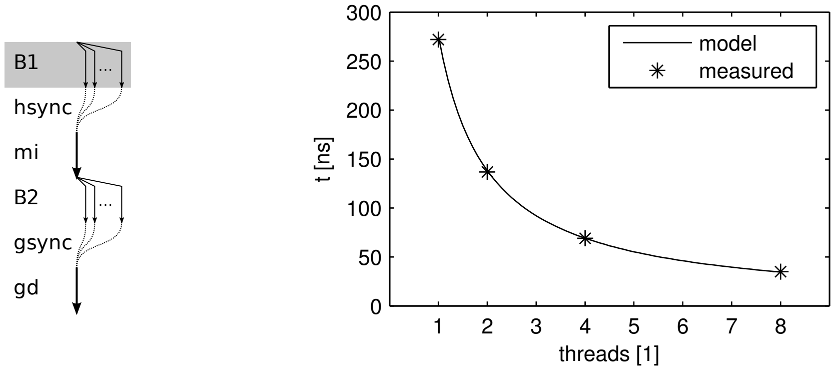


Fig. 6.4: t_{B1} per point pair and iteration for different N_T

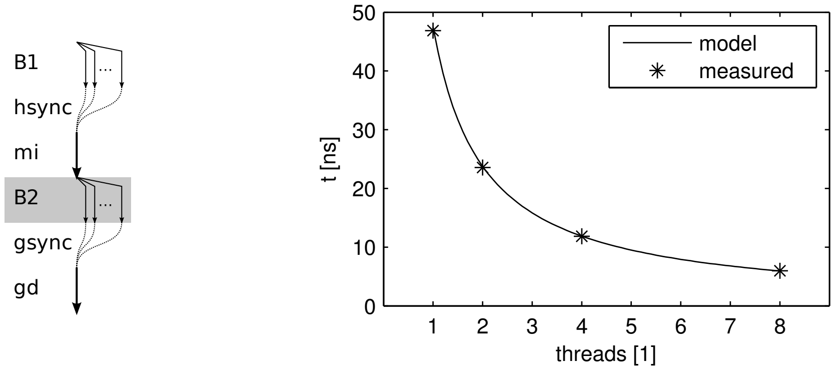


Fig. 6.5: t_{B2} per point pair and iteration for different N_T

First, we estimate the parameters of the model functions for the parallelized stages. These model functions were defined in section 5.5.1. Based on the measurements, the expressions

$$t_{B1}\left(N_P, N_I, N_T\right) = 276.3\frac{N_P N_I}{N_T}\ \mathrm{ns} \tag{6.3}$$

$$t_{B2}\left(N_P, N_I, N_T\right) = 47.5\frac{N_P N_I}{N_T}\ \mathrm{ns} \tag{6.4}$$

were derived using least-squares fitting. Figures 6.4 and 6.5 show the measurements and the model functions for t_{B1} and t_{B2}.

An important condition for these model functions to hold is that with increasing N_T enough memory bandwidth is provided such that the threads do not suffer from starvation. Otherwise these stages become memory bound. The memory bandwidth requirements will be discussed in section 6.4. We will see that these stages of the algorithm are compute bound on the Cell/B.E. processor and that the performance is not limited by the memory bandwidth.

In these two stages, there is almost no synchronization or communication overhead. The only communication that takes place is a start message sent by the PPE to each SPE and a completion message sent back. Latency of such messages is in the order of microseconds [5] and was not detectable in the measurements. Because N_P is in the order of millions, t_{B1} (equation 6.3) and t_{B2} (equation 6.4) are in the order of tens to hundreds of milliseconds.

Moreover, additional experiments with different knot spacings, 8 and 32 voxel units respectively, showed that N_K has no significant impact on t_{B1} and t_{B2} as long as we assume that the knot spacing is greater than the image block width. Otherwise, the grouping-based code optimizations become less efficient.

6.2.3 Communication Overhead

The parallel algorithm introduces some additional overhead related to the collection of the partial results of the accelerator cores. After the execution of phase B1, the sum of all partial histograms is calculated. We defined the model function

$$\begin{aligned} t_{hsync}\left(N_B, N_I, N_T\right) = \\ \left(c_{hsync,1} + c_{hsync,2}N_T + \left(c_{hsync,3} + c_{hsync,4}N_T\right)N_B\right)N_I, \end{aligned} \tag{6.5}$$

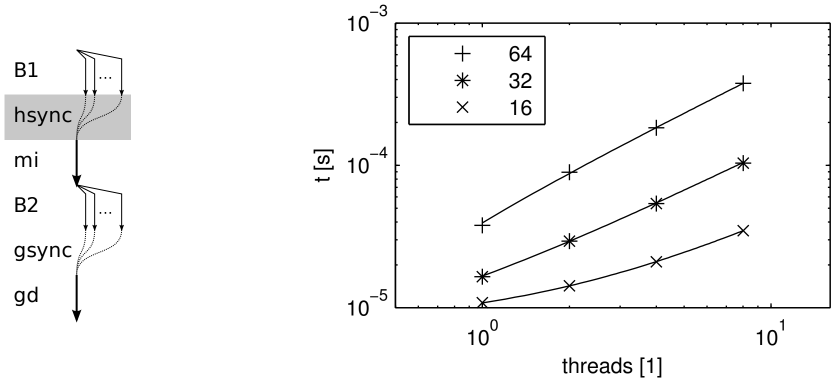


Fig. 6.6: Comparing measurements (marks) and the model function (lines) for t_{hsync} for different histogram sizes (16×19, 32×35 and 64×67 bins).

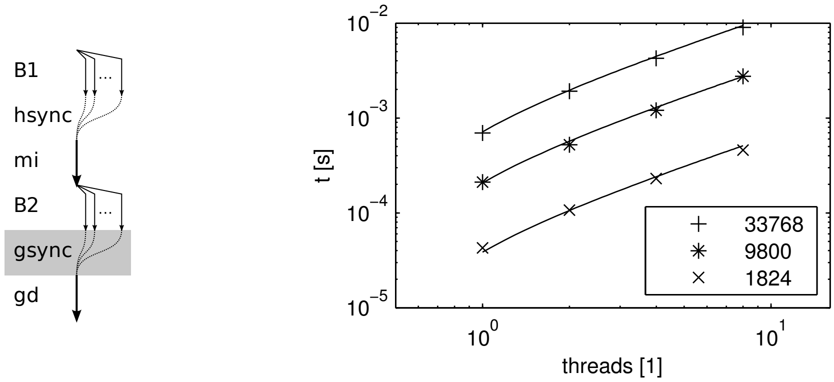


Fig. 6.7: Comparing measurements (marks) and the model function (lines) for t_{gsync} for different grid sizes (1824, 9800 and 33768 knots).

for this communication phase. When estimating the parameters for the model function using the measurements, the formula becomes

$$t_{hsync}\left(N_B, N_I, N_T\right) = \\ \big(8.63 \cdot 10^3 + 5.37 N_T + \left(-4.04 + 11.23 N_T\right) N_B\big) N_I \text{ ns}. \tag{6.6}$$

The processing of the histogram of the first SPE is cheaper because it only has to be transferred to main memory, while all the others have to be transferred and then added. This is reflected by the negative parameter $c_{hsync,3}$.

The runtime of t_{gsync} is modeled by a similar function. However, the signaling overhead is smaller because the partial gradient results are already in main memory at the start of this phase. Unlike for the histogram synchronization, no message command from the PPE to all SPEs to initiate the transfer from local store to main memory is required. The size of the gradient data structure is generally larger than the histogram. Therefore, the message-length independent term $c_{gsync,1} + c_{gsync,2} N_T$ is expected to be relatively small compared to the message-length dependent term $\left(c_{gsync,3} + c_{gsync,4} N_T\right) N_K$. The model is therefore simplified by omitting the message-length independent term. Estimating the parameters based on the measurements with the 22 datasets gives

$$t_{gsync}\left(N_K, N_I, N_T\right) = \left(-15.95 + 37.03 N_T\right) N_K N_I \text{ ns}. \tag{6.7}$$

Figures 6.6 and 6.7 show model predictions and measurements for t_{hsync} and t_{gsync} for different histogram and transformation grid sizes respectively.

The values for $c_{gsync,3}$ and $c_{gsync,4}$ are roughly four times larger than the ones for $c_{hsync,3}$ and $c_{hsync,4}$ because the gradient contains a vector for each knot, while the histogram stores one scalar value per bin.

6.2.4 The Sequential Part

There are two sequential phases of the algorithm, running exclusively on the PPE. One is the computation of the joint and marginal pdfs and the mutual information based on the joint histogram. The total runtime is approximately proportional to the bin count.

$$t_{mi}\left(N_B, N_I\right) = 505.38 N_B N_I \text{ ns} \tag{6.8}$$

Figure 6.8 shows model and measurements of t_{mi} for different histogram sizes and N_T. Especially for low bin counts, the model is not very accurate, probably because there is a small sequential part, which is not proportional to N_B. The model for t_{gd} derived from the measurements is

$$t_{gd}\left(N_K, N_I\right) = 123.91 N_K N_I \text{ ns.} \tag{6.9}$$

The runtime is proportional to N_K and independent of N_T. Figure 6.9 shows model and measurements. For small N_K the measured time can be significantly lower than the estimate possibly due to cache effects, which compromises the accuracy of the model. The gradient descent optimizer uses the gradient array, which is written in the preceding stage. Therefore, the cache hit ratio may be high especially in the case where the gradient array is smaller than the cache size of the PPE.

These two functions are not inherently sequential. It would be possible to parallelize them, but it was a design decision based on the performance measurements of the sequential algorithm to not offload them to the worker threads because the possible performance benefit is minimal.

Finally there are the parts of the algorithm that are not assigned to any of the other stages. They consume around 130 microseconds per iteration.

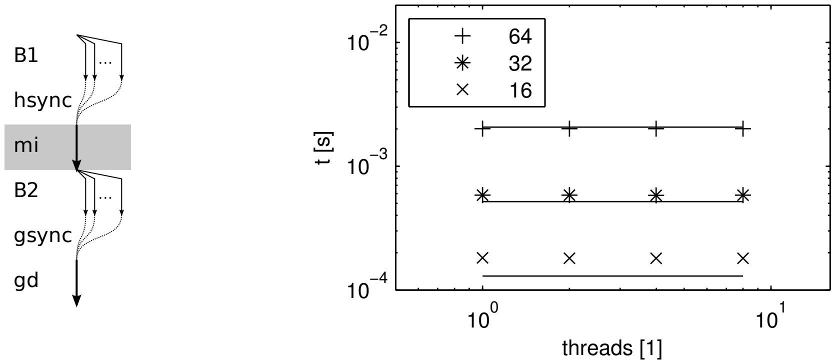


Fig. 6.8: Comparing measurements (marks) and the model function for t_{mi} per iteration for different histogram sizes (16×19, 32×35, 64×67 bins)

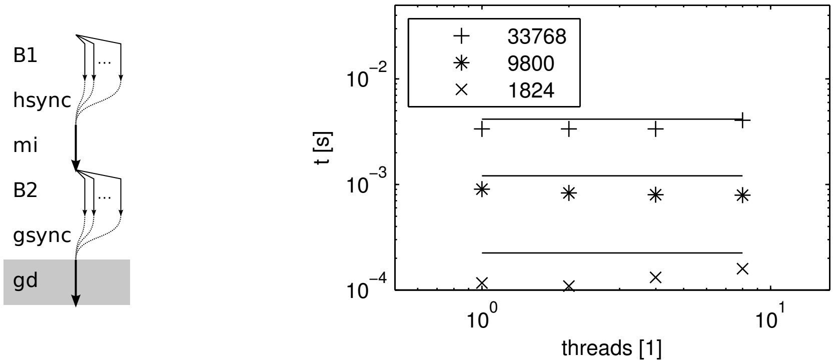


Fig. 6.9: Comparing measurements (marks) and the model function for t_{gd} per iteration for different grid sizes (1824, 9800 and 33768 knots).

6.3 Scalability 1: Many-Core

Based on the model established in the previous section, the performance of the parallel algorithm on a next generation Cell/B.E. with more SPEs or on a similar architecture with many cores can be estimated and possible problems can be identified. Subject of interest in this section is the overhead due to thread synchronization (t_{gsync} and t_{hsync}). Another important issue, the memory bandwidth requirements of the calculation parts B1 and B2 will be discussed in the next section.

6.3.1 Fixed Problem Size

A way to look at scalability is related to what is known as Amdahl's law, which bases on a statement made by Gene Amdahl in 1967 to argue against the usefulness of massively parallel architectures [7]. He observed that algorithms can be divided into a sequential and a parallelizable part and that for many problems the sequential part is a considerable fraction of the total runtime and can already dominate the algorithm performance for moderate levels of parallelism.

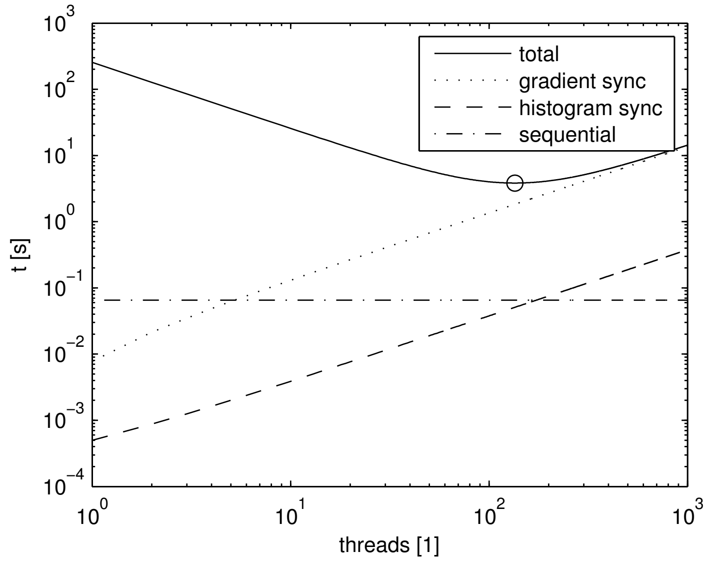


Fig. 6.10: Scalability for a fixed problem size. For large N_T, the overhead induced by the collection of the partial results of the threads becomes the bottleneck

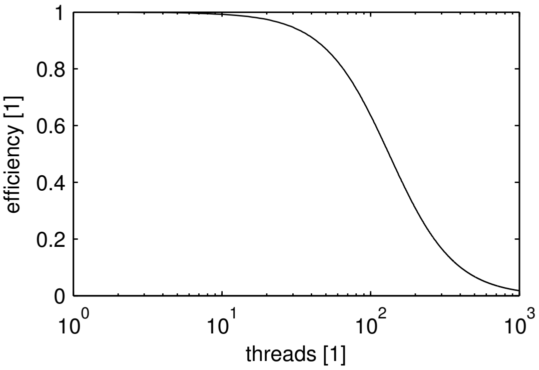


Fig. 6.11: The efficiency decreases for large N_T. For $N_T \geq 43$ it is lower than 0.9

In our case, Amdahl's law suggests that beyond a certain N_T, adding more cores does not result in significant performance gain. We look at the dependence of the runtime on N_T for an image size of $512 \times 512 \times 100$ voxels, a grid point spacing of 16 voxels, 32×35 histogram bins and $N_I = 30$.

The lowest runtime reached for these settings is 3.8s for $N_T = 135$ (Figure 6.10). This corresponds to a speedup of 66×. For greater N_T, the runtime even starts to increase again. For $N_T \geq 43$, the efficiency is below 0.9 (Figure 6.11).

The reason why the algorithm does not scale beyond 135 worker threads is not a dominating sequential part but an increasing overhead for the collection of the partial results of the worker threads: t_{gsync} (and t_{hsync}) increase linearly with N_T. While this is no problem on one Cell/B.E. processor, where t_{gsync} is about 0.3% of the total runtime, it possibly becomes the bottleneck on future systems with many processor cores. On such systems, the way how the global histogram and gradient are obtained from the partial results calculated by the worker threads has to be reconsidered. In the following sections, ways to efficiently implement these tasks for large N_T will be discussed.

6.3.2 Reducing Gradient Collection Overhead

We recall that t_{gsync} increases linearly with N_T because the master thread (on the PPE) has to calculate the sum of the N_T partial gradients produced by the worker threads. The reason why each thread has its private copy of the gradient is that it allows to avoid locking overhead if multiple threads possibly write to the same location of the gradient. This strategy may be sufficiently efficient for small N_T but for large N_T it is too expensive and a different solution is needed. The availability of efficient atomic update functions, for example, could eliminate this problem, but we will focus on adaptations of the current scheme for large N_T. Atomic functions allow to perform a read-modify-write operation atomically, eliminating the possibility of interference with another thread.

Parallelization

An approach to reducing t_{gsync} is to parallelize this stage and offload it to the worker threads. If $N_K \geq N_T$, we can partition the transformation grid into N_T domains and assign each domain to a worker thread, which then collects the partial results within this domain. Such parallelization ideally results in an additional factor $\frac{1}{N_T}$ in (6.7), yielding

$$t_{gsync,2}\left(N_K, N_I, N_T\right) = \frac{1}{N_T}\left(-15.95 + 37.03 N_T\right) N_K N_I \mathrm{ns}. \tag{6.10}$$

and an asymptotic behavior like

$$\lim_{N_T \to \infty} t_{gsync,2}\left(N_K, N_I, N_T\right) = 37.03 N_K N_I \mathrm{ns}, \tag{6.11}$$

meaning that $t_{gsync,2}$ becomes independent of N_T.

In reality, the SPE implementation performs even better because manual vectorization and double-buffering were implemented. For arrays with 2^{16} entries, which is roughly the size of the gradient for the chosen problem size, a significant reduction of t_{gsync} can be achieved (Figure 6.12). However, the parallelization is not ideal, since we can still observe a dependency on the number of threads. This is at least partially related to memory bandwidth. This operation is likely to be memory bound because it does little computation on a relatively large amount of data. The aggregate bandwidth saturates just above 20 GByte/s (Figure 6.13), which is close to the theoretical 25.6 GByte/s bandwidth cap to external memory. It is difficult to predict the behavior on a future system because it will heavily depend on the memory bandwidth. However, by replacing (6.7) with

$$t_{gsync} = \frac{\left(N_T + 1\right) N_K \times 16\mathrm{Byte}}{BW}, \tag{6.12}$$

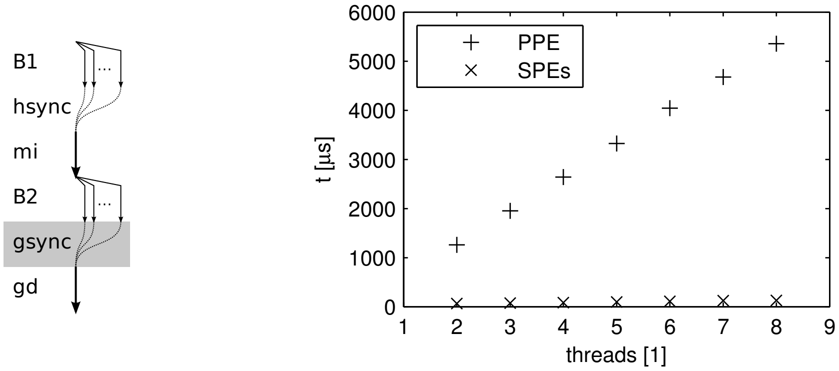


Fig. 6.12: Time required to calculate the sum of N_T arrays with 2^{16} entries. For $N_T = 8$, a reduction from $5353\mu\mathrm{s}$ to $127.5\mu\mathrm{s}$ can be achieved by offloading this task to the SPEs. This is more than $8\times$ because vectorization and double buffering were implemented for the SPE.

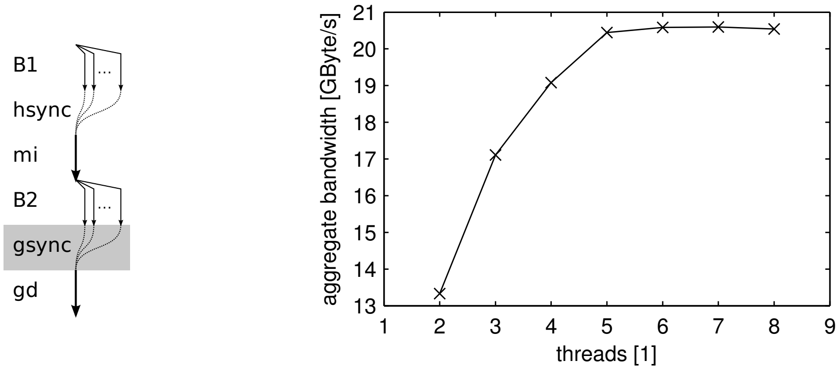


Fig. 6.13: The time required to calculate the sum of large arrays on the SPEs is memory bound. Saturation is reached at roughly 20 GByte/s for 5 SPEs with an array size of 2^{22}

where BW is the aggregate bandwidth to memory, the memory boundness of the operation is taken into account. There is a factor $N_T + 1$ because N_T arrays are read and one array is written.

Gradient Sharing

Although the solution discussed in the preceding section partially alleviates the problem, there is still room for improvement regarding memory bandwidth requirements. Because the domain decomposition is based on the fixed image space, neighboring domains will overlap with respect to the transformation grid knots in their sphere of influence. With increasing N_T, the domains become smaller and there are more and more domains without overlap. Threads associated to domains without overlap can work on the same copy of the gradient without requiring locking mechanisms because they will never try to modify the same data. Sharing a copy of the gradient among multiple threads results in a total number of gradient copies N_{PG} less than N_T and instead of (6.12), we get

$$t_{gsync} = \frac{(N_{PG} + 1)\, N_K \times 16\text{Byte}}{BW} \times N_I. \tag{6.13}$$

The problem of finding N_{PG} can be expressed as a graph-coloring problem, where the threads are the nodes of the graph and nodes are connected by edges if there is at least one grid point which is modified by both threads. N_{PG} is the chromatic number of the graph-coloring problem. Finding the exact chromatic number is an NP-complete problem [49] and finding the best coloring may be too time consuming. In practice, heuristics are often used to find a sufficiently good solution for NP-complete problems in a reasonable time. For the chosen problem size, the number of copies of the gradient can be reduced significantly with a simple greedy heuristic. This heuristic creates a gradient copy for the first thread. For each of the following threads it checks the available gradient copies one after another for overlaps with its assigned threads. If a gradient copy with no conflicts is found, the current thread is assigned to it, otherwise a new gradient copy is created for the current thread. For the chosen problem size, this allows to reduce the number of copies of the gradient to less than 10% for large N_T.

The experiments have shown that t_{B1} and t_{B2} do not depend on the knot spacing N_K. A conclusion of (6.13) is that this also roughly holds for t_{gsync}. If we assume an arbitrarily large image, a knot spacing of one image block width and one thread per image block, N_{PG} is $4 \times 4 \times 4 = 64$. If we double the knot spacing, N_{PG} becomes $8 \times 8 \times 8 = 512$. When neglecting the effects caused by the B-spline padding, N_K gets reduced by the same factor as N_{PG} increases. Therefore, the product $(N_{PG} + 1)N_K$ in (6.13) remains approximately constant. A finer transformation grid increases the size of the gradient array, but it reduces overlaps and makes sharing among threads easier. A coarser grid yields smaller gradient arrays but makes sharing more difficult because the overlaps become bigger.

Prediction

Because t_{gsync} is bandwidth limited, it may be difficult to predict the performance of this stage on future systems. But figure 6.14 shows that the estimated t_{gsync} based on (6.13), under the very conservative assumption that this operation will also saturate at $BW = 20$ GByte/s on future systems, ceases to be the bottleneck. However, t_{hsync} may now limit performance.

6.3.3 Reducing Histogram Collection Overhead

The time t_{hsync} grows linearly with the number of threads N_T. An approach similar to gradient sharing as described previously is not possible because write access of a thread cannot be restricted to a part of the histogram only. A parallelized implementation on the SPEs, however, is possible. One could implement a scheme like for the parallel gradient collection. This would require two stages. First the PPE has to instruct all the SPEs to transfer their histograms to main memory. When all transfers are finished, the PPE sends a command message to the SPEs to start calculating the sum of the partial results, each SPE taking care of a subset of the bins.

Collective Communication

The fact that the histograms are small and already located in the local store and not in main memory favors a collective reduce operation. In this collective communication operation, the cost of collecting the partial histograms increases logarithmically with N_T.

Figure 6.15 shows the collective communication pattern of the histogram calculation on one Cell/B.E. processor. When SPE1 finishes the calculation of its partial histogram, it notifies SPE0 of this event. SPE0 after finishing the calculation of its partial histogram waits for the notification from SPE1

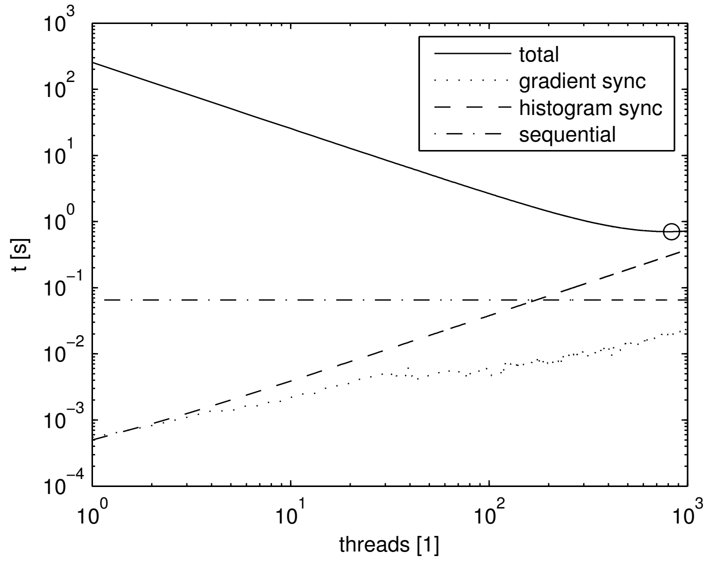


Fig. 6.14: If threads working on non-overlapping domains share their private copy of the gradient and parallelism is exploited to calculate the sum of the private copies, t_{gsync} can be reduced significantly even if the memory bandwidth would be limited to 20 GByte/s on future systems. t_{hsync} now becomes the bottleneck for large N_T. We can observe an irregular behavior of the plot of t_{gsync}. The reason is that for some N_T the heuristic is more successful in reducing the number of gradient copies than for others.

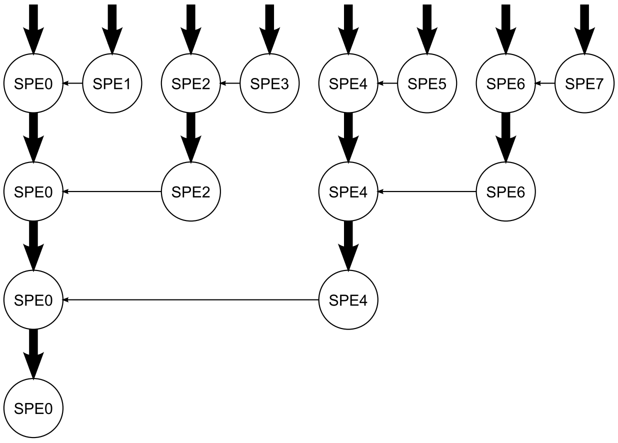


Fig. 6.15: The SPEs pair-wise calculate the sum of their partial histograms. The vertical arrows are computation phases: first the computation the partial histograms and then the pair-wise calculation of the sum of the partial histograms. The horizontal arrows stand for pair-wise synchronization and data transfer. Finally, the histogram is available on SPE0, which transfers it to main memory and notifies completion to the PPE. The reduce operation is initiated directly after finishing the calculation of the partial histograms and does not have to be triggered by the PPE.

and then initiates an LS-to-LS DMA transfer to fetch the partial histogram of SPE1 and adds it to its partial histogram. After repeating this notify-fetch-add pattern with SPE2 and SPE4, the final histogram is in the local store of SPE0, which then transfers it to main memory and notifies the PPE.

The model function for this communication pattern is

$$t_{hsync} = (c_1 + c_2 N_B + \log_2 N_T \times (c_3 + c_4 N_B))\, N_I\, \mu s. \tag{6.14}$$

The first two parameters c_1 and c_2 account for messages and DMA transfers between the PPE and the SPEs. The tree structure of the communication pattern has $\log_2 N_T$ levels. In each level, there are operations independent of the number of bins N_B as well as operations that depend on N_B. The parameters c_3 and c_4 are used to model the runtime of these per-level tasks.

Because the SPEs only synchronize pair-wise at each level of the reduction tree, it is, for example, possible that SPE2 sends its notification to SPE0 before SPE1. It therefore makes sense to use signaling notification with the signal-notification register in OR mode. If SPE2 and SPE1 set a different bit in the signal-notification register, the order in which the notifications are received by SPE0 does not matter. SPE-to-SPE signal notification is faster than PPE-to-SPE message passing.

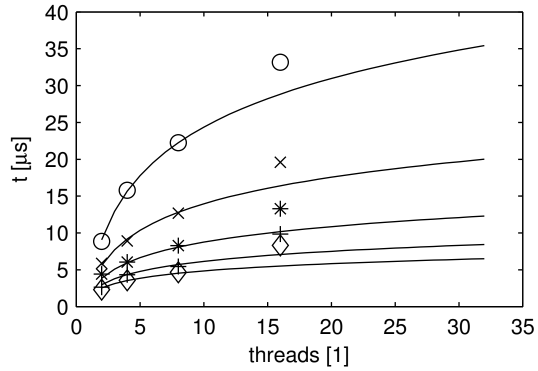


Fig. 6.16: Time required to calculate the sum of N_T histograms of different sizes (2^8, 2^9, 2^{10}, 2^{11} and 2^{12} bins). For $N_T > 8$, SPEs on different chips are involved. This makes the operation more expensive, because the signals have to cross chip-boundaries.

Figure 6.16 shows t_{hsync} per iteration for different N_T and N_B. The model function

$$t_{hsync} = \big(1.50 + 0.24 \cdot 10^{-3} N_B + \log_2 N_T \times \big(0.62 + 1.46 \cdot 10^{-3} N_B\big)\big)\, N_I\ \mu s \tag{6.15}$$

was derived ignoring the data measured for $N_T = 16$, which was acquired using the two processors on a QS21. This data point was included in the figure to show that there is an additional latency penalty of about $3\mu s$ if SPEs on different chips have to be synchronized. The signals used by the SPEs to communicate have a higher latency if chip boundaries are crossed. Therefore, it is again difficult to predict the behavior of future architectures because they are likely to show asymmetries. Probably, similar to today's large-scale distributed computing systems, the choice of the optimal implementation of the reduce function will depend on the interconnect topology [29]. High-end many-core chips may even use hardware assists to minimize the latency of such collective communication operations, as it is common for today's high-end distributed systems [6].

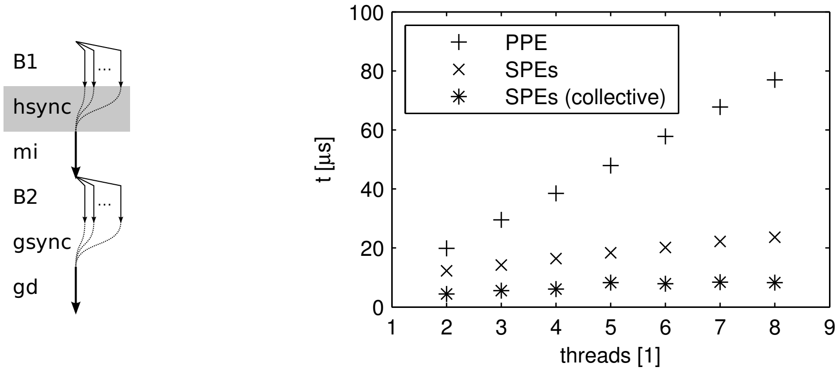


Fig. 6.17: Time required to calculate the sum of N_T arrays with 2^{10} entries. Although offloading to the SPEs alone improves the performance already, the signaling overhead significantly contributes to the total runtime. With the collective communication approach, this overhead is reduced and a logarithmic dependency on N_T can be achieved.

Figure 6.17 shows that a parallel implementation similar to the one described in section 6.3.2 is already much more efficient than calculation on the PPE. A collective communication approach increases efficiency further, as it removes the requirement of having one central point for all the synchronization by having the SPEs exchange messages among themselves.

Prediction

The updated model predictions (Figure 6.18) demonstrate that the calculation of the sum of the partial histograms as a collective communication operation on the SPEs should be sufficiently efficient also on chips with many cores, such that neither t_{gsync} nor t_{hsync} limit the performance of the algorithm and a good efficiency can be achieved on many-core processors (Figure 6.19).

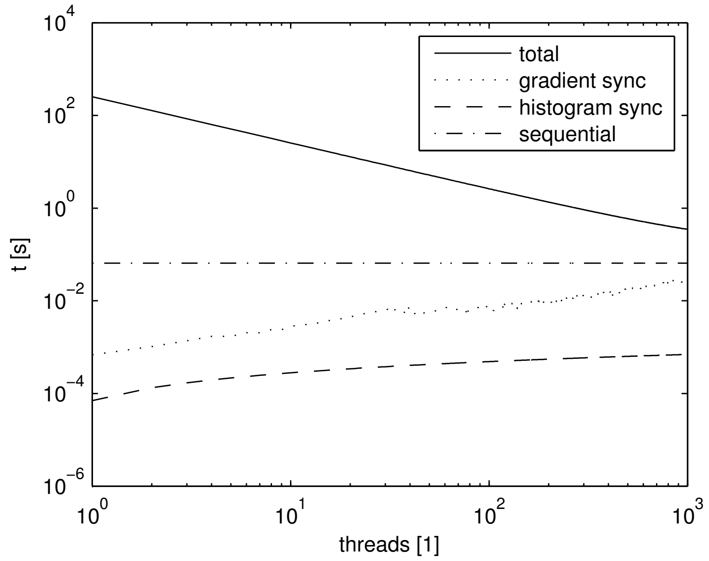


Fig. 6.18: Estimated runtime for a fixed problem size on a hypothetical many-core chip with optimized gradient and histogram synchronization.

6.3.4 Scaling Problem Size

So far, a fixed problem size with $N_P = 512 \times 512 \times 100$ was assumed. As stated by Amdahl's law, with an increasing number of threads, the efficiency decreases because the sequential part becomes more dominant (Figure 6.19). The flaw of this argument is that it assumes a fixed problem size. John L. Gustafson showed later that massively parallel systems indeed have their justification because they allow to process larger problems in the same time as a smaller system would require to process a smaller problem [37]. He argued that given a more powerful system, the problem generally

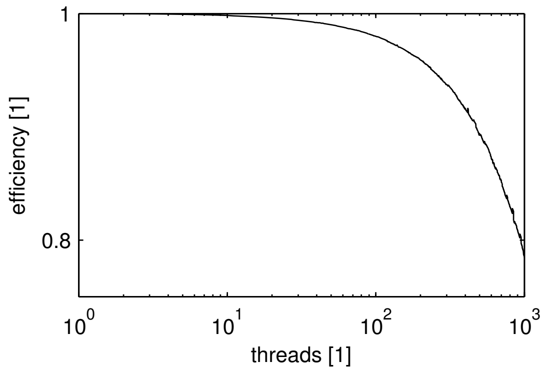


Fig. 6.19: Estimated efficiency for a fixed problem size on a hypothetical many-core chip with optimized gradient and histogram synchronization.

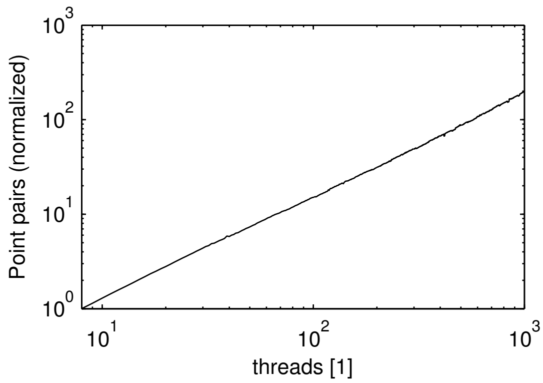


Fig. 6.20: The isoefficiency curve for $E = 0.9981$ (the efficiency for $N_P = 512 \times 512 \times 100 =$ 26'214'400 and $N_T = 8$) is approximately linear and thus indicates good scalability of the algorithm. The y-axis shows the problem size N_P as a multiple of 26'214'400.

is expanded to make use of the additional processing power. In our case, this means that the number of point pairs N_P increases because the images are acquired with a higher resolution. The isoefficiency analysis by Kumar et al. [55] looks at scalability from this perspective. If the problem size N_P needs to grow as $f(N_T)$ to maintain an efficiency E, $f(N_T)$ is the isoefficiency function. Figure 6.20 shows the isoefficiency curve for $E = 0.9981$, the efficiency for $N_T = 8$ and $N_P = 512 \times 512 \times 100$, based on the optimized communication schemes discussed in the preceding sections. The observed linear isoefficiency curve indicates good scalability. An exponential isoefficiency curve, for example, would mean that it is difficult to utilize parallelism for this algorithm and architecture combination for large N_T because the required problem sizes become huge.

6.3.5 Conclusions

So far we have discussed the scalability of the algorithm on multicore (or many-core) systems based on a subdivision in parallel, sequential and communication tasks. The initial communication scheme developed for the Cell/B.E. processor with its 8 SPEs showed limited scalability in the case of large numbers of cores. Some adaptations were necessary for such systems and we observed that runtime for some stages is not bound by computation power any more but by memory bandwidth. In the next section, we will analyze under what conditions this can also become the case for the main computation phases, and therefore for t_{B1} and t_{B2}.

6.4 Memory Considerations

6.4.1 The Memory Wall

In 1994, Wulf and McKee coined the term “memory wall” [106] for the observation that as technology evolves, more and more workloads will not be bound by the peak performance of the microprocessor but how fast it can be fed with data. There is a gap between the rates with which microprocessor and memory speed improved in the past. Although both developments are exponential, in the case of the microprocessor development the exponent is substantially larger, causing the gap to increase over time. In 2004, McKee observed that there are already many applications with a performance limited by the memory bottleneck, while others keep improving in performance with improvements in processor speeds [69]. In this section, we want to analyze how close the parallel nonrigid registration algorithm is to the “memory wall”.

6.4.2 Bandwidth Requirements

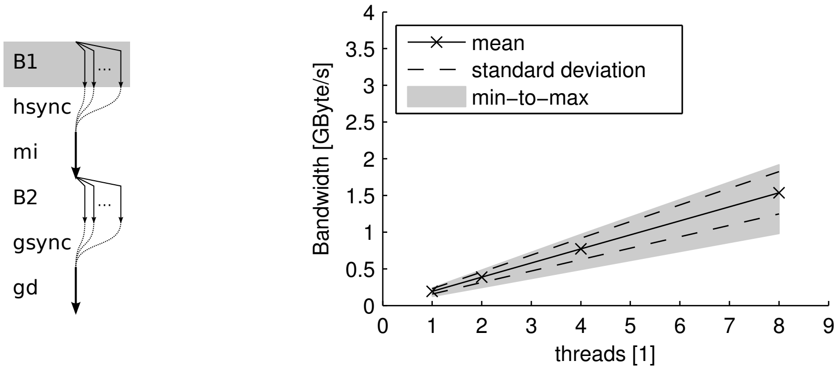


Fig. 6.21: The bandwidth requirements are problem dependent for the histogram calculation. The shaded region covers the minimum to the maximum value over the 22 image pairs. The theoretical peak memory bandwidth of the Cell/B.E. processor is 25.6 GByte/s.

While some of the data intensive stages of the algorithm are memory bound already today (section 6.3.2), the two most time consuming ones are still compute bound. The bandwidth requirements for the histogram calculation depends on the actual registration problem, as the number of

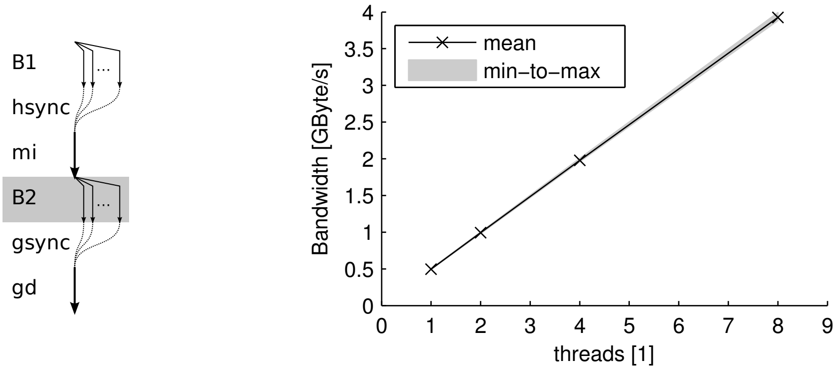


Fig. 6.22: The bandwidth requirements are not problem dependent for the gradient calculation. The (almost invisible) shaded region covers the minimum to the maximum value over the 22 image pairs. The theoretical peak memory bandwidth of the Cell/B.E. processor is 25.6 GByte/s.

moving image blocks that overlap with a given fixed image block changes with the resulting transformation function and the sampling rate of the fixed and moving image. This does not apply for the gradient calculation if temporary results are stored, where bandwidth requirements are problem independent. Figures 6.21 and 6.22 show the memory traffic on the Cell/B.E. (without prefetching), which reaches 4 GByte/s during gradient calculation. It is below the theoretical peak bandwidth of 25.6 GByte/s. This may change for future processors if available bandwidth keeps lagging behind processor development. General purpose processors generally have a lower memory bandwidth than the Cell/B.E. architecture. For the Core2 processor, we measured a copy bandwidth of 5.56 GByte/s using the STREAM benchmark [68]. There is no probabilistic prefetching in phase B2. Therefore, we can assume that the amount of data transferred during this phase is similar on both platforms. Considering that the runtime is less than 1.5× higher on the Core2, the memory traffic may come close to 3 GByte/s and doubling the number of cores possibly causes the memory bandwidth to become the bottleneck.

If computational power is abundant and memory bandwidth BW scarce, we can reformulate the equations (6.3) and (6.4) to the form

$$t_{B1,bw}\left(N_P, N_I, BW\right) = k_{B1}\frac{N_P N_I}{BW}\ \text{ns} \tag{6.16}$$

$$t_{B2,bw}\left(N_P, N_I, BW\right) = k_{B2}\frac{N_P N_I}{BW}\ \text{ns} \tag{6.17}$$

with constants k_{B1} and k_{B2} to be determined. By filling in the the measured average bandwidths for one worker thread (198 MByte/s for phase B1 and 508 MByte/s for phase B2) and solving $t_{B1,bw} = t_{B1}$ and $t_{B2,bw} = t_{B2}$ for k_{B1} and k_{B2}, we get

$$k_{B1} = 53.41 GByte/s \tag{6.18}$$

$$k_{B2} = 23.58 GByte/s. \tag{6.19}$$

These constants can be interpreted as the bandwidth required to process one point pair per nanosecond in an iteration of the respective phase.

6.4.3 Prefetching

The performance measurements in section 6.1 show that double-buffering and prefetching of moving image data only have a minor impact on the runtime of the algorithm. This is surprising considering that double-buffering is a well-known optimization technique on the Cell/B.E architecture.

First, we want to take a closer look at prefetching. The parameter d_{thr} is used to control how many moving image blocks are prefetched. Figure 6.23 shows the memory traffic during the histogram calculation as a function of d_{thr}. Even for large d_{thr}, the bandwidth is well below the theoretical limit of 25.6 GByte/s. A large d_{thr} therefore has no negative impact on performance on the Cell/B.E. The goal of prefetching is reducing the miss rate — optimally to zero — for moving image data reads.

Figure 6.24 shows the dependency of moving image data misses on d_{thr} for one of the 22 image pairs. For $d_{thr} = 0$, no moving image blocks are prefetched. This results in almost 4 moving image blocks per fixed image block being fetched during the histogram calculation, each of them causing the SPE to stall until the DMA transfer is finished. When increasing d_{thr}, moving image blocks start to get prefetched, reducing the miss rate. For $d_{thr} > 1.2$, some moving image blocks that are not required start to get fetched, which increases the total number of fetched blocks. For $d_{thr} = 1.5$,

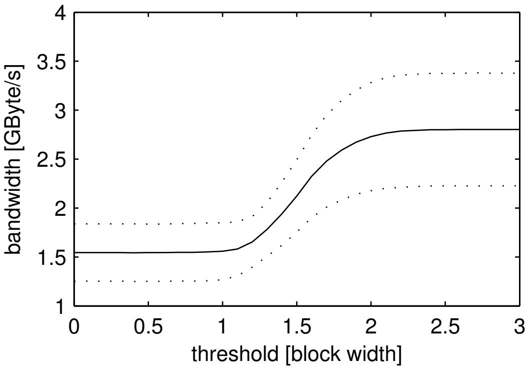


Fig. 6.23: The average memory traffic (over all 22 image pairs) during histogram calculation depends on the prefetch threshold d_{thr}. A larger d_{thr} means a more aggressive prefetching of moving image blocks and therefore more memory traffic. The dotted lines indicate the standard deviation.

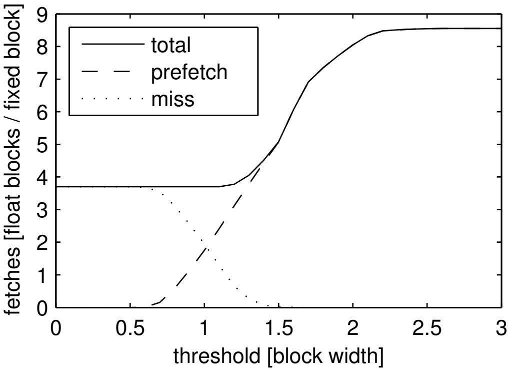


Fig. 6.24: With increasing d_{thr} the number of fetches caused by cache misses decreases and the number of prefetched blocks increases. However, the total number of fetched blocks also increases for larger d_{thr}, meaning that some of the prefetched blocks are not required.

the miss rate is reduced to 0 without causing a too excessive overhead. Thus, 1.5 would be a good value for d_{thr}. By further increasing d_{thr}, more blocks that are not required get transferred until the curve plateaus between 8 and 9 moving image blocks being fetched per fixed image block.

This shows that prefetching allows to reduce the cache miss ratio without too much increase in memory traffic. However, in order to understand the impact on performance, we also need to know the cache miss penalty.

6.4.4 Cache Miss Penalty

The penalty attributed to a stall due to missing moving image data is the time required to fetch a moving image block. The duration of such a DMA transfer depends on the transfer size and the physical location of the SPE on which we measure the latency (Figure 6.25). SPE0 is on the chip to which the memory containing the data is attached while SPE8 is on the other chip. The first interesting observation is that fetching one Kilobyte of data takes about the same time as fetching 16 Bytes and the penalty starts to rise only for larger blocks. The cache miss penalties are in the order of hundreds of nanoseconds: 300ns (466ns on SPE8) for a fixed image block of 2 kByte and about double that in the case of the moving image blocks, which are larger due to the replicated data.

Loading the transformation field coefficients is a different case. They have to be fetched in 16 small pieces because only coefficients that are neighbors in x-direction are stored in adjacent memory locations. However, these transfers can be in parallel — we can issue the DMA commands one after another without waiting for the previous one to finish — and it takes around 1.2μs to fetch the entire set of coefficients related to one fixed image block.

The time to process one block in the histogram calculation stage is $t_{B1}(512, 1, 1) = 141.45\mu$s (see eq. 6.3). The time required to fetch a block of data is much smaller: in the hundreds of nanoseconds. Double buffering and prefetching have no significant performance impact because only a very small number of transfers are necessary during the processing of one block (the fixed image block, around 4 moving image blocks and the transformation coefficients).

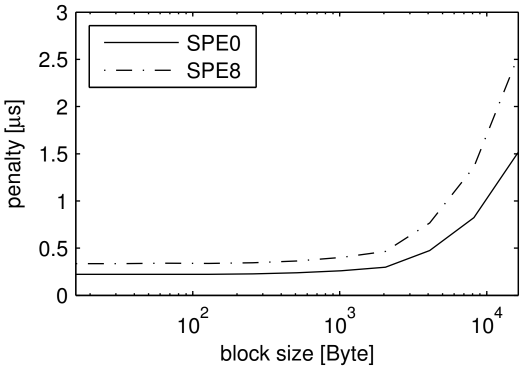


Fig. 6.25: The duration of a DMA transfer as a function of the amount of data transferred was measured on a QS21. The data is stored in the memory attached to the chip containing SPE0, while SPE8 is on the other chip and therefore sees a higher latency to memory. The operating system decides where the memory for the moving image data is allocated. For small block sizes, the time required for the DMA transfer is almost constant. When the block size becomes larger than about 2 kByte, the latency starts to rise. In each experiment, only one SPE was active and the memory bus bandwidth did not saturate.

6.4.5 Conclusions

The Cell/B.E. processor has a theoretical memory bandwidth limit of 25.6 GByte/s. The main computational stages of the parallel nonrigid registration algorithm peak at around 4 GByte/s. This suggests that the algorithm will remain compute bound at least in the near future.

The data required to process one fixed image block is transferred to the local store with relatively few, large DMA transfers. This small number of fetches is a result of the restructuring of the algorithm and its data structures inspired by the Cell/B.E. memory architecture. The explicit management of the cache encourages the programmer to maximize data locality and fetch data in large blocks. While these paradigms are especially important on streaming processors like the Cell/B.E., they are also beneficial for the hardware prefetch units of a general purpose processor (see section 6.1).

The double-buffering of data does not have a significant impact on the performance. However, hiding the memory latency may become more important in the future. Similar to the disparity observed between processing power and memory bandwidth, bandwidth improves more rapidly over time than latency. David A. Patterson gives the rule of thumb: “*In the time that bandwidth doubles, latency improves by no more than a factor of 1.2 to 1.4.*” and notes that it is often easy to improve bandwidth at the expense of latency [80]. Moreover, latencies may increase if ever more cores issue long DMA transfers on one shared bus.

6.5 Scalability 2: Cell/B.E. Cluster

The preceding sections dealt with a performance and scalability analysis on (future) multicore processors beyond the 8 SPEs on one Cell/B.E. processor. A system with more SPEs can already be built today by connecting multiple QS21 Cell/B.E. blades. Each blade contains two processors and a Gigabit Ethernet adapter. By combining four such nodes, we get a system with a total of 64 SPEs. On such a system, we have additional communication overhead (see section 5.7). In the following sections, we first measure the additional communication overhead, extend the performance model using the results and repeat the scalability analysis for the updated model. Finally, the runtime for the 22 image pairs already used earlier is measured on the Cell/B.E. cluster.

6.5.1 Communication Cost on a QS21 Cluster

As discussed in section 5.7.2, synchronization of the master threads is necessary. An additional `MPI_Allreduce` operation is necessary after each of the two synchronization stages.

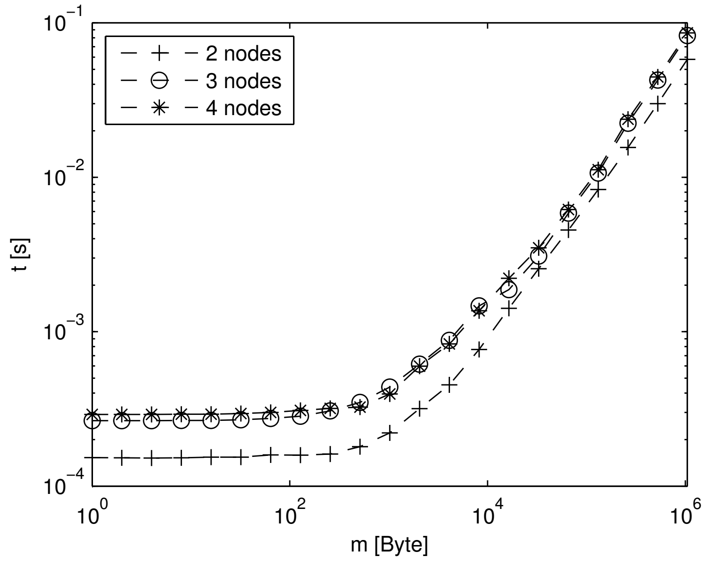


Fig. 6.26: Time required for an `MPI_Allreduce` operation depending on message size and node count

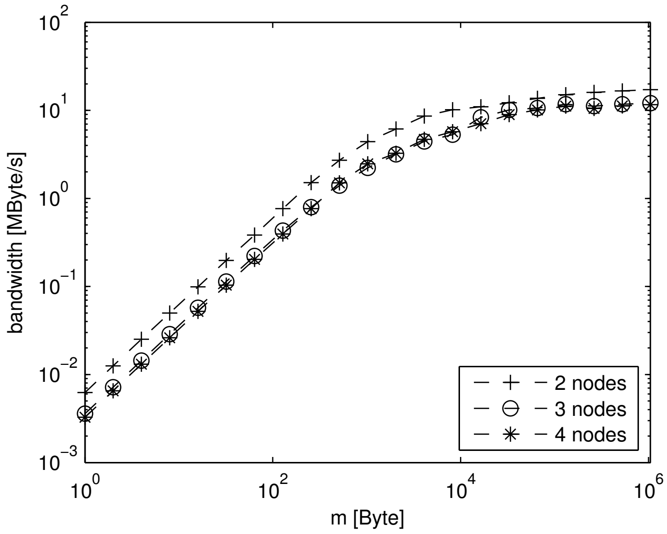


Fig. 6.27: Bandwidth for an `MPI_Allreduce` operation depending on message size and node count

On the cluster consisting of four QS21 blades, the performance of an `MPI_Allreduce` operation for different message sizes and numbers of nodes using *OpenMPI* over Ethernet was measured. Figure 6.26 shows the geometric mean values calculated from 25 measurements for each data point and figure 6.27 the resulting bandwidth. The pseudo code for the function used to acquire one data point is

Listing 6.1: Measuring the runtime of an `MPI_Allreduce` operation

```
nloop = 1
do
  MPI_Barrier
  MPI_Allreduce //warm-up
  start = MPI_Wtime()
  for j = 1 to nloop
    MPI_Allreduce
  end
  time = MPI_Wtime() - start
  nloop *= 2
while (time < 1s)
nloop /= 2
return time/nloop
```

For small message sizes, multiple reductions are carried out sequentially so that the total time measured is at least one second. We can observe that the range where $t_0\left(n\right)$ dominates (5.20) goes little beyond $m = 100$ Byte, while for larger message sizes $r_\infty\left(n\right)$ becomes dominant. The latency for $n = 2$ is 152μs and rises by 139μs to 291μs if n is doubled. The asymptotic bandwidth is 17.21MByte/s for $n = 2$. It is reduced to 11.34MByte/s if n is doubled. These values should not be confused with the latency and bandwidth of the network because multiple transfers between different nodes are necessary during the `MPI_Allreduce` operation. The latency observed in our system is very high considering that some supercomputing interconnects reach sub-10μs point-to-point latencies [62]. This problem was already observed on similar clusters by Bolten et al. but they demonstrate that significant improvement is possible by using InfiniBand interconnects [12]. The bandwidth that we observe, however, is comparable to what Chou et al. [16] observed for a different Gigabit Ethernet based clusters.

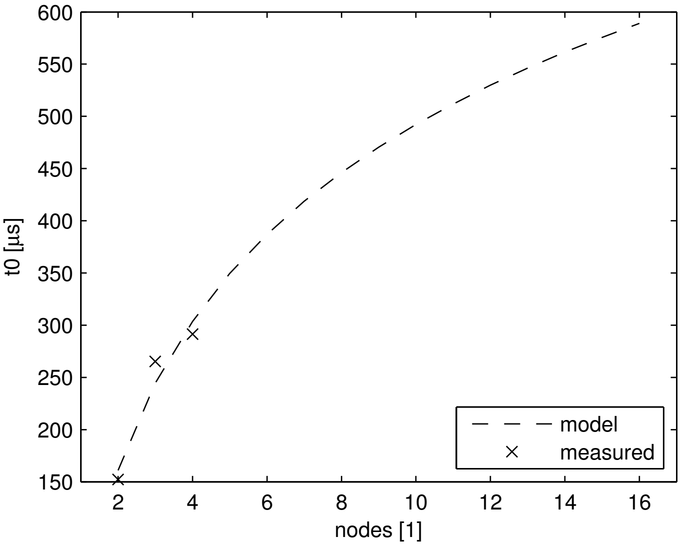


Fig. 6.28: Measured and modeled t_0 for QS21 Ethernet clusters

If we use the data points for $m \leq 16$ Byte to obtain the latency related parameters c and d and the data points for $m \geq 256$kByte to estimate the bandwidth related parameters e and f of the model function (5.23) defined in section 5.7.2, we get

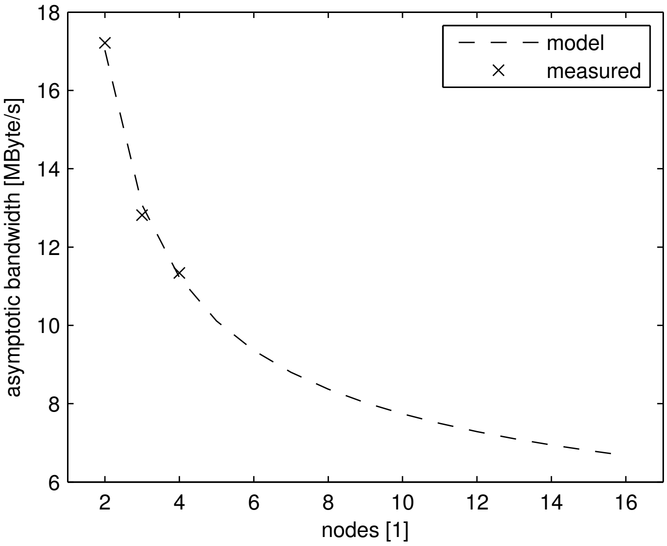


Fig. 6.29: Measured and modeled asymptotic bandwidth for QS21 Ethernet clusters

$$t_{\text{Allreduce}}(n,m) = (142.71\log_2 n + 18.18)\,\mu\text{s}+ \\ (28.96\log_2 n + 27.05)\,\frac{\text{ns}}{\text{Byte}}m \tag{6.20}$$

We distinguish between the latency dependent part

$$t_{lat,\text{Allreduce}}(n,m) = (142.71\log_2 n + 18.18)\,\mu\text{s} \tag{6.21}$$

and the bandwidth dependent part

$$t_{bw,\text{Allreduce}}(n,m) = (28.96\log_2 n + 27.05)\,\frac{\text{ns}}{\text{Byte}}m \tag{6.22}$$

The predictions and measurements of the asymptotic bandwidth and latency for different node counts are in figures 6.28 and 6.29 respectively. Measurements and predictions of the cost of an `MPI_Allreduce` operation with $m = 1$MByte are shown in figure 6.30.

We recall that for our workload we have to synchronize the histogram and the gradient in every iteration. A histogram of 32 × 32 bins consisting of single precision floats occupies 4kByte. The gradient on a grid of

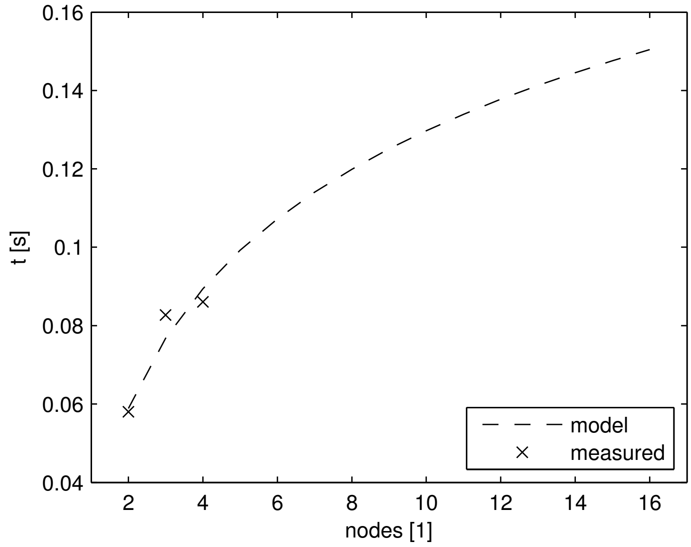


Fig. 6.30: Measured and modeled time for an `MPI_Allreduce` operation depending on the node count (m = 1 MB)

$16 \times 16 \times 16$ knots occupies 64kByte if the values are stored as single precision float vectors. Using (6.20), the estimated synchronization cost per iteration for the histogram is 0.39 ms on a system with two nodes and 1.18 ms on a system with 16 nodes. For the gradient, the estimates are 3.90 ms and 9.33 ms respectively, giving a total overhead of 4.29 ms for 2 nodes and 10.51 ms for 16 nodes. The major contribution comes from the bandwidth dependent part: 4.14 ms and 9.88 ms respectively. For our workload, the asymptotic bandwidth of the interconnect is therefore the more important parameter than the latency. Using a low-latency interconnect is not expected to increase the performance for our workload significantly.

6.5.2 Extending the Model

With the parameters for the inter-node communication cost derived in the preceding section, all the parameters of the model functions for the synchronization cost on the Cell/B.E. cluster are known. These functions, each containing a term for the inter-node and one for the intra-node synchronization, were defined in section 5.7.2 and are repeated here:

$$t_{hsync,cluster} = t_{hsync}\left(N_B, N_I, N_{T,l}\right) + t_{\text{Allreduce}}\left(N_N, N_B \times 4\text{Byte}\right) \qquad (6.23)$$

$$t_{gsync,cluster} = t_{gsync}\left(N_K, N_I, N_{T,l}\right) + t_{\text{Allreduce}}\left(N_N, N_K \times 16\text{Byte}\right). \quad (6.24)$$

Figure 6.31 shows the estimated performance depending on the cluster size (assuming one processor per node) for a fixed image size of $512 \times 512 \times 100$ voxels. The x-axis covers a range up to 1024 processors, meaning 128 nodes. This is a very large system considering that the model functions were derived based on a much smaller system consisting of four nodes and designing the interconnect network for such a large system may not be trivial. However, it allows to qualitatively compare the graph with the many-core model derived earlier (Figure 6.18). As expected, communication overhead starts to limit the performance already for smaller N_T than in the many-core case. An efficiency greater than 0.9 can be achieved only for systems with up to four nodes (Figure 6.32) for this problem size. Again, the problem size would have to be increased to achieve a better efficiency on a larger system.

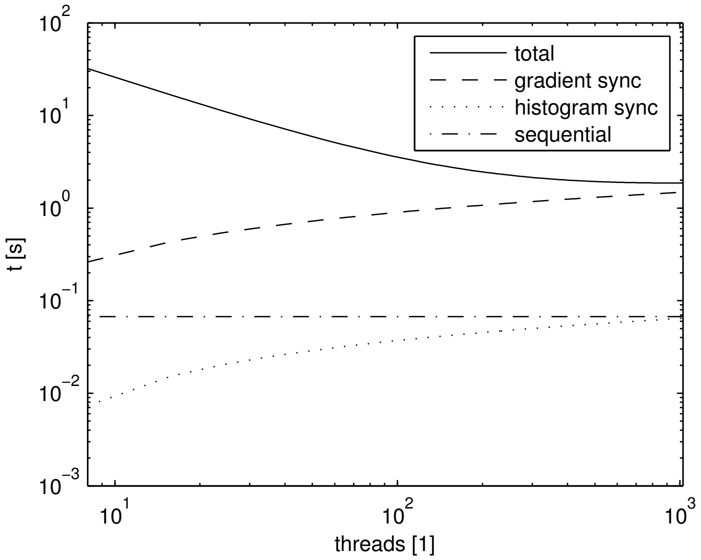


Fig. 6.31: Scalability on a cluster with 1 Cell/B.E. processor per cluster node for a fixed image with $512 \times 512 \times 100$ voxels.

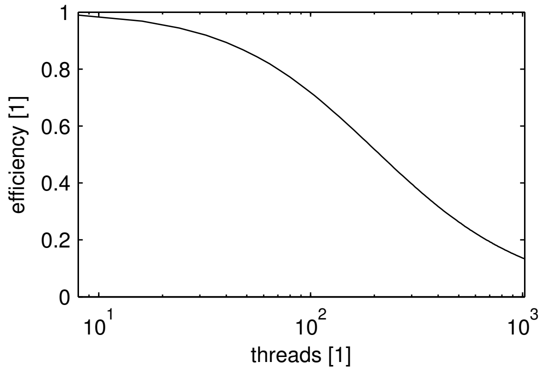


Fig. 6.32: Efficiency on a cluster with 1 Cell/B.E. processor per cluster node for a fixed image with $512 \times 512 \times 100$ voxels.

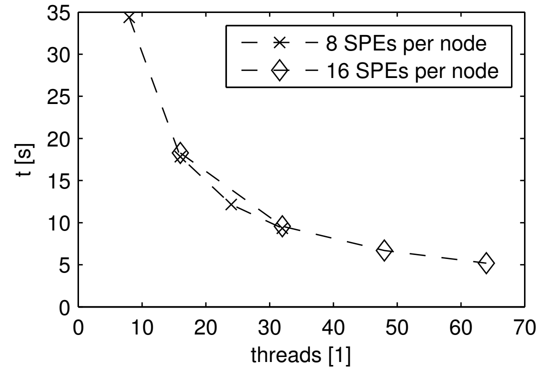


Fig. 6.33: The measured runtime for a fixed image size of $512 \times 512 \times 98$ voxels for different numbers of nodes and different numbers of enabled processors per node.

6.5.3 Measurements on the QS21 Cluster

The runtime of the registration of the 22 image pairs used for the benchmark (section 6.1) was measured on the Cell/B.E. cluster. For a sample image with a size of $512 \times 512 \times 98$ voxels, the runtime was 5.18s on the Cell/B.E. cluster (Figure 6.33). The average runtime per voxel and iteration was 6.7164ns. This corresponds to an efficiency of 0.78 compared to the single-threaded implementation.

7

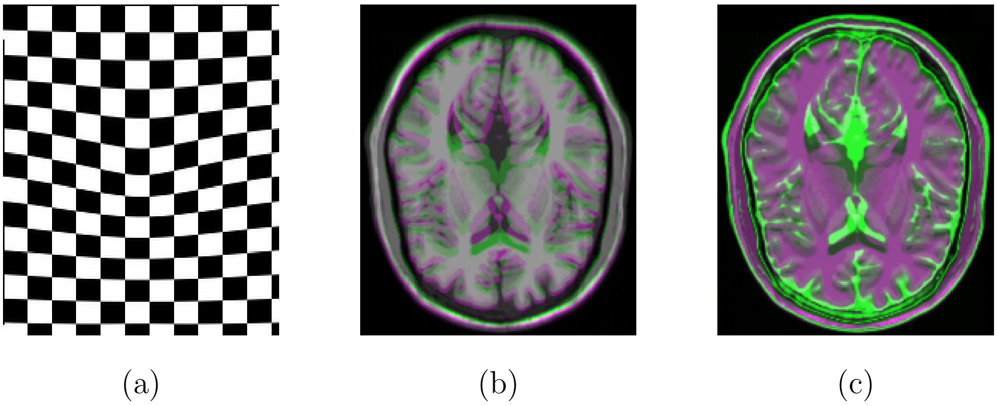
Experiments

In the preceding chapter, the performance and scalability of the streaming algorithm were analyzed. For this purpose, a fixed number of iterations was executed for different image pairs. In the following experiments, automatic detection of convergence is enabled. If the metric has not improved by more than a certain threshold value over the last five successful steps of the gradient descent optimizer, the algorithm stops optimizing.

Furthermore, multiresolution is enabled. The images are scaled by a factor s^l, where l is the image-pyramid level and s is the scale factor between two levels. The transformation grid is scaled together with the images. The grid spacing is the same in all levels in terms of voxel units. The result of the registration at one resolution level is used to initialize the transformation at the next level, moving towards higher resolutions. For this purpose, the transformation needs to be resampled. One QS21 Cell/B.E. blade will be used for the experiments.

7.1 Simulated MR Images

The simulated BrainWeb images, which were already introduced in section 2.1, are used for this experiment. Their size is 181 × 217 × 181 voxels with a voxel spacing of 1 mm. We generate a smooth transformation based



(a) (b) (c)

Fig. 7.1: A slice of (a) the warped checkerboard image, (b) $I_{T1,w}$ (red and blue channel) and I_{T1} (green channel), and (c) $I_{T1,w}$ (red and blue channel) and I_{T2} (green channel).

on thin plate splines[1], which yields a maximum displacement of 20mm at the image center. Figure 7.1a shows a slice of a checkerboard image, to which the transformation was applied. The two aligned BrainWeb images of different modalities, I_{T1} and I_{T2}, are deformed with this transformation function to obtain the warped images $I_{T1,w}$ and $I_{T2,w}$. The registration of I_{T1} (moving image) to $I_{T1,w}$ (fixed image) is a monomodal registration problem and the registration of I_{T2} (moving image) to $I_{T1,w}$ (fixed image) is a multimodal one. A fusion of the fixed and the moving image shows the misalignment for the monomodal (Figure 7.1b) and the multimodal case (Figure 7.1c). In the monomodal case and for perfect alignment, such a fusion would yield a grayscale image. This is not the case for the multimodal problem because corresponding points generally do not have the same intensity.

The registration parameters were:

- Transformation grid spacing of 16 voxel units
- 32 fixed and $32 + 3$ moving image bins
- Four multiresolution levels ($l = 0 \dots 3$)
- Scale factor of 0.7 between levels ($s = 0.7$)

---

[1] Thin plate splines are another widely used transformation model. Their name refers to the physical analogy of bending a thin piece of metal.

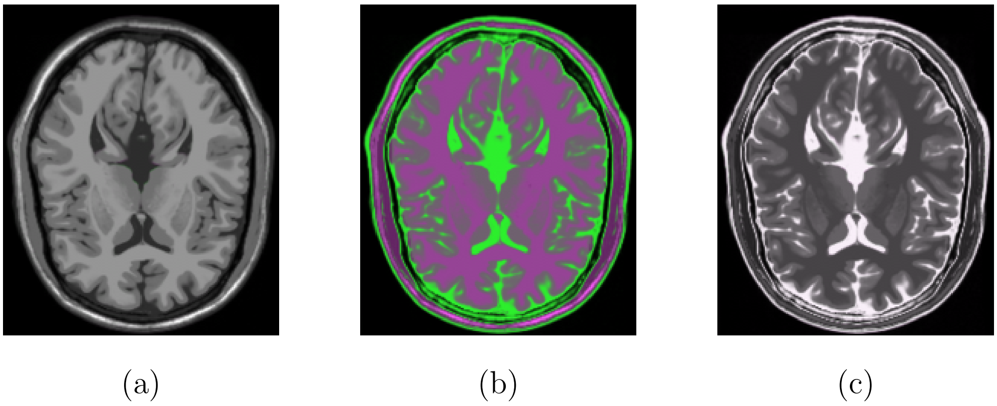
	runtime [s]	
	monomodal	multimodal
File I/O	0.18	0.17
Image Pyramid	0.97	0.95
Transformation Pyramid	0.05	0.05
Registration	4.65	4.20
Total	**5.85**	**5.36**

Tab. 7.1: The runtime of monomodal and multimodal registration on one QS21 is around 5 to 6 seconds. In addition to the actual registration, this includes the time required to load the files from disk and to construct the image and transformation pyramids.

In the final level, convergence is reached after 17 iterations for the monomodal registration and after 15 iterations in the multimodal case. The total runtimes are around five to six seconds (Table 7.1). Additional to the actual registration, there is some time required to load the files from disk, including rearrangement from row-major to block-8 mapping and type casting. For this example, this takes less than 0.2 seconds. Furthermore, for the multiresolution pyramid, the images need to be resampled and the B-spline representations of the moving images needs to be computed. In total, this takes almost a second in these experiments. Depending on the application, the pyramid of one of the images may be computed offline, for example the preoperative image in intraoperative registration or the atlas image in the case of atlas-based segmentation. Resampling the transformation function between the pyramid levels takes only little time, around 0.05 seconds overall in these experiments.

Visual inspection of the warped and registered images shows that both registrations were successful. Figure 7.2a shows the result of the monomodal case. It is the result of the registration problem in figure 7.1b. Figure 7.2b shows the result of the multimodal example. It is the result of the registration problem in figure 7.1c. Here, the accuracy of the registration is more difficult to inspect because the corresponding points in the two images in general do not have the same intensity and therefore the fusion of two aligned images is not gray. However, fusing the result of the registration with the warped image $I_{T2,w}$ shows the success of the registration.

We also measured the displacement at all points except the ones belonging to the background (more than 4 million points). Before registration,



(a) (b) (c)

Fig. 7.2: A slice of (a) $I_{T1,w}$ (red and blue channel) and $I_{T1,r}$ (green channel), and (b) $I_{T1,w}$ (red and blue channel) and $I_{T2,r}$ (green channel), and (c) $I_{T2,w}$ (red and blue channel) and $I_{T2,r}$ (green channel). $I_{T1,w}$ was the fixed image for both registrations. Image (c) shows that the multimodal registration was indeed successful, which may be difficult to see in image (b).

the average displacement was 4.53 mm. After registration, it was 0.06 mm in the monomodal and 0.04 mm in the multimodal case. The voxel size is one millimeter and thus the registration accuracy is comparable to the "constantly better than 0.1 pixels" reported by Kybic and Unser [57] for a similar, monomodal experiment.

7.2 CT Images

In this experiment, two abdominal CT images are registered. They are from the same patient but were acquired at different times. The fixed and moving image have 512 × 512 × 98 and 512 × 512 × 94 voxels respectively. Prior to nonrigid registration, the images were rigidly registered (Figure 7.3a) with the algorithm described by Ohara et al. [75].

The same settings as for the preceding experiments are used for the nonrigid registration. In the final level, convergence is reached after 23 iterations. The total runtime is below 30 seconds (Table 7.2). While the construction of the image pyramid requires around three seconds, the resampling of the transformation function is very fast compared to the rest of the algorithm.

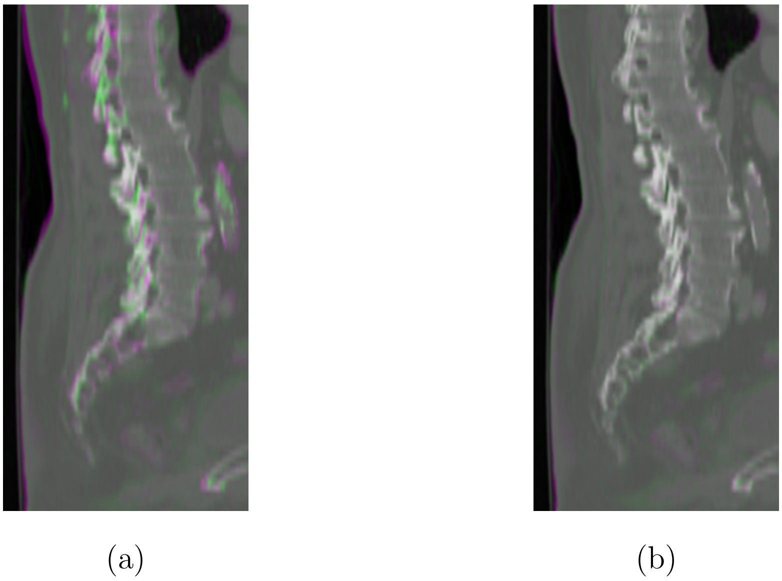

(a) (b)

Fig. 7.3: Fusion of the fixed image (red and blue channel) with the (a) rigidly registered moving image (green channel) and (b) nonrigidly registered moving image (green channel).

Registration of abdominal images may be very challenging because of the large number of structures present and the variation of their shape and position over time. The spine, for instance, is an articulated body. In this case, nonrigid registration improved its mapping compared to rigid registration (Figure 7.3).

7.3 Comparison with Other Algorithms

A truly objective comparison of the runtime of nonrigid registration algorithms is always difficult because of the differences in the algorithms and the experimental setups. The image size, for example, has a significant impact on the registration time. We use the runtime per voxel to compare our algorithm to other parallel registration algorithms.

Taking the results from the last section and omitting the time to load the images, a registration time of 0.96μs/voxel (24.72 s for $512 \times 512 \times 98$ voxels) is achieved. Rohlfing et al. [85] and Ino et al. [47] presented multimodal nonrigid registration algorithms running on much larger systems. They

	runtime [s]
File I/O	0.75
Image Pyramid	3.15
Transformation Pyramid	0.10
Registration	21.47
Total	**25.48**

Tab. 7.2: The runtime of the nonrigid registration of the two CT images is below 30 seconds. In addition to the actual registration, this includes the time required for loading the files from disk and constructing the image and transformation pyramids.

report a normalized runtime of 17.9μs/voxel (4.7 minutes for $512 \times 512 \times 60$ voxels) on a 64 processor system and 11.5μs/voxel (8 minutes for $512 \times 512 \times 159$ voxels) on a 128 processor system respectively.

An FPGA-based multimodal nonrigid registration algorithm by Dandekar et al. [27] requires 21.5μs/voxel (6 minutes for 256^3 voxels). We are not aware of any multimodal 3D nonrigid registration algorithm for GPUs. The fastest monomodal algorithm we are aware of was presented by Muyan-Özçelik et al. [72] with a runtime of 1.09μs/voxel (12.46 s for $424 \times 180 \times 150$ voxels).

8

Conclusions and Outlook

Nonrigid registration is important for the comparison of biomedical images which were acquired with different devices, at different times or from different patients. It aligns the images and therefore allows their fusion. Nonrigid registration of two 3D images typically takes several hours, which prohibits interactive use. This is a major obstacle for a widespread adoption in clinical environments and faster solutions are especially required in intraoperative scenarios with their strict time constraints.

There are implementations reaching registration times in the order of minutes, but they typically base on relatively large multiprocessor architectures. The cost of acquisition and maintenance of such systems limits the availability of these solutions. The algorithm presented in this dissertation achieves sub-minute registration on relatively small and low-cost systems. Compared to an open-source implementation running sequentially on a general-purpose processor of the same generation as the Cell/B.E. processor, a speedup of more than 2500× was achieved. Optimizations of the algorithm and its data structures yielded a speedup of almost 80× and moving to a Cell/B.E. system with two processors added another 34× improvement. The Cell/B.E. processor, which is designed for multimedia and gaming applications, consists of a general purpose processor and eight accelerator cores specialized in single-precision floating-point calculations. The algorithm models the nonrigid transformations with B-splines, which is a widespread approach. The degrees of freedom of the model can be adapted to the respective registration problem. A mutual-information-based similarity metric is utilized. It enables multimodal registration, but can also be applied to monomodal registration problems.

Standard CPUs also have become multicore processors and the trend seems to go towards an increasing number of cores rather than more performant individual cores. This strategy enables a higher performance per chip area (or Watt). Our parallel algorithm and implementation can directly benefit from current microprocessor development trends. The performance difference of the used processors is relatively small compared to the speedup achieved by optimizing and restructuring the algorithm.

A scalability analysis showed that the algorithm can continue to scale well with an increasing number of cores at least in the near future. An efficient memory access pattern and prefetching help avoiding problems related to the memory wall and large memory latencies. The analysis was carried out on the Cell/B.E. platform, which is bandwidth optimized. To achieve a high throughput, data need to be transferred in large blocks. Standard processors have a different memory hierarchy and typically a lower memory bandwidth than the Cell/B.E. architecture. Furthermore, memory performance increases slower than computational performance. It remains to be seen if memory bandwidth will become the limiting factor on general purpose multicore processors. It may become necessary to reconsider certain design decisions which are increasing memory traffic. For instance, some of the moving image data at the borders of image blocks were replicated in order to enable an efficient loading to vector registers. Furthermore, some intermediate results were stored for reuse in later stages of the algorithm.

Graphics processing units can provide high memory bandwith and excellent computational performance. The spectrum of applications for which they are suitable becomes wider, not only because the architectures become more generic but also due to improvements in the programming environments. While fast GPU-based monomodal nonrigid registration has already been demonstrated by a number of groups, the data-dependent histogram access pattern during multimodal registration still poses a problem to GPUs. More recent designs include a writable on-chip memory, enabling efficient histogram computation. However, the size of this memory is still too small for the relatively large bin count of a typical joint histogram. Nevertheless, GPU-based acceleration of multimodal nonrigid image registration may become feasible in the near future and offer the most cost-effective solution to this problem. For this, a specific investigation of the performance and bottlenecks, and appropriate algorithmic adjustments will be necessary, analog to the methods presented in this thesis.

A

List of abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
Cell/B.E.	Cell Broadband Engine
Cg	C for graphics
CMOS	Complementary Metal-Oxide-Semiconductor
CPI	Cycles per Instruction
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
EIB	Element Interconnect Bus
FEM	Finite Element Method
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPGPU	General-Purpose computation on Graphics Processing Units
GPU	Graphics Processing Unit
ITK	Insight Toolkit
LS	Local Store

LUT	Lookup Table
MFC	Memory Flow Controller
MPI	Message Passing Interface
MR	Magnetic Resonance
MRI	Magnetic Resonance Imaging
Mutex	Mutual Exclusion
NUMA	Non-Uniform Memory Access
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PDE	Partial Differential Equation
pdf	probability density function
POSIX	Portable Operating System Interface
PPE	Power Processor Element
PPU	Power Processor Unit
Pthreads	POSIX threads
RISC	Reduced Instruction Set Computer
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
SPE	Synergistic Processor Element
SPU	Synergistic Processor Unit
SSD	Sum of Squared Differences
SSE	Streaming SIMD Extensions
TCP	Transmission Control Protocol
UMA	Uniform Memory Access
VLIW	Very Long Instruction Word
VMX	Vector Multimedia Extensions
XML	Extensible Markup Language

Bibliography

[1] GNU Compiler Collection. `http://gcc.gnu.org/`.

[2] ITK - Segmentation & Registration Toolkit. `http://www.itk.org/`.

[3] OpenMPI. `http://www.open-mpi.org/`.

[4] Oprofile. `http://oprofile.sourceforge.net/`.

[5] J. L. Abellán, J. Fernández, and M. E. Acacio. Characterizing the basic synchronization and communication operations in dual cell-based blades. In *Proc. ICCS, Part I*, pages 456–465, 2008.

[6] G. Almási, G. Dózsa, C. C. Erway, and B. D. Steinmacher-Burow. Efficient implementation of allreduce on BlueGene/L collective network. In *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 57–66. Springer, 2005.

[7] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proc. AFIPS*, pages 483–485, 1967.

[8] A. Andronache, P. C. Cattin, and G. Székely. Adaptive subdivision for hierarchical non-rigid registration of multi-modal images using mutual information. In *Proc. MICCAI*, volume 2, pages 976–983, 2005.

[9] A. Arevalo, R. M. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas, and C. Almond. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Redbooks. Vervante, 2008.

[10] R. Bajcsy and S. Kovačič. Multiresolution elastic matching. *Comput. Vision Graph. Image Process.*, 46(1):1–21, 1989.

[11] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proc. SC*, pages 1–11, 2008.

[12] M. Bolten, A. Dolfen, N. Eicker, I. Gutheil, W. Homberg, E. Koch, A. Schiller, G. Sutmann, and L. Yang. JUICE - Jülich initiative Cell cluster report 2007. Technical report, Jülich Supercomputing Centre, 2007.

[13] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[14] M. Bro-Nielsen and C. Gramkow. Fast fluid registration of medical images. In *Proc. VBC*, pages 267–276, 1996.

[15] L. G. Brown. A survey of image registration techniques. *ACM Comput. Surv.*, 24(4):325–376, 1992.

[16] C.-Y. Chou, H.-Y. Chang, S.-T. Wang, and S.-C. Tcheng. Modeling message-passing overhead on NCHC Formosa PC cluster. In *Proc. GPC*, Lecture Notes in Computer Science, pages 299–307, 2006.

[17] N. Chrisochoides, A. Fedorov, A. Kot, N. Archip, P. Black, O. Clatz, A. Golby, R. Kikinis, and S. Warfield. Toward real-time, image guided neurosurgery using distributed and grid computing. In *SuperComputing06*, Tampa, Florida, USA, 2006.

[18] G. E. Christensen, M. I. Miller, M. W. Vannier, and U. Grenander. Individualizing neuroanatomical atlases using a massively parallel computer. *Computer*, 29(1):32–38, 1996.

[19] G. E. Christensen, R. D. Rabbitt, and M. I. Miller. Deformable templates using large deformation kinematics. *IEEE Trans. Image Process.*, 5(10):1435–1447, 1996.

[20] U. Clarenz, M. Droske, and M. Rumpf. Towards fast non-rigid registration. In *Inverse Problems, Image Analysis and Medical Imaging, AMS Special Session Interaction of Inverse Problems and Image Analysis*, pages 67–84. AMS, 2002.

[21] W. E. Cohen. Tuning programs with OProfile. *Wide Open Magazine*, pages 53–62, 2004.

[22] D. L. Collins, A. P. Zijdenbos, V. Kollokian, J. G. Sled, N. J. Kabani, C. J. Holmes, and A. C. Evans. Design and construction of a realistic digital brain phantom. *IEEE Trans. Med. Imag.*, 17(3):463–468, 1998.

[23] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[24] N. Courty and P. Hellier. Accelerating 3D Non-Rigid Registration using Graphics Hardware. *International Journal of Image and Graphics*, 8:1–18, 2008.

[25] W. R. Crum, T. Hartkens, and D. L. Hill. Non-rigid image registration: theory and practice. *The British Journal of Radiology*, 77(2):140–153, 2004.

[26] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., 1999.

[27] O. Dandekar and R. Shekhar. FPGA-accelerated deformable image registration for improved target-delineation during CT-guided interventions. *IEEE Transactions on Biomedical Circuits and Systems*, 1(2):116–127, 2007.

[28] J. Doweck. Inside Intel core microarchitecture and smart memory access. Technical report, Intel Corporation, 2006.

[29] A. Faraj and X. Yuan. An empirical approach for efficient all-to-all personalized communication on ethernet switched clusters. In *Proc. ICPP*, pages 321–328, 2005.

[30] M. Ferrant, A. Nabavi, B. Macq, P. M. Black, F. A. Jolesz, R. Kikinis, and S. K. Warfield. Serial registration of intraoperative MR images of the brain. *Medical Image Analysis*, 6:337–359, 2002.

[31] A. F. Frangi, D. Rueckert, J. A. Schnabel, and W. Niessen. Automatic construction of multiple-object three-dimensional statistical shape models: Application to cardiac modeling. *IEEE Trans. Med. Imag.*, 21(9):1151–1166, 2002.

[32] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing.* Addison Wesley, second edition, 2003.

[33] S. Green. Image processing tricks in OpenGL. In *Game Developers's Conference 2005*, 2005.

[34] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[35] A. Guimond, A. Roche, N. Ayache, and J. Meunier. Three-dimensional multimodal brain warping using the demons algorithm and adaptive intensity corrections. *IEEE Trans. Med. Imag.*, 20(1):58–69, 2001.

[36] T. S. Gunawan and W. Cai. Performance analysis of a myrinet-based cluster. *Cluster Computing*, 6(4):299–313, 2003.

[37] J. L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[38] A. Hagemann, K. Rohr, H. Stiehl, U. Spetzger, and J. Gilsbach. Biomechanical modeling of the human head for physically-based, non-rigid image registration. *IEEE Trans. Med. Imag.*, 18(10):875–884, 1999.

[39] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. Technical report, Intel Corporation, 2006.

[40] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., third edition, 2003.

[41] D. L. G. Hill, P. G. Batchelor, M. Holden, and D. J. Hawkes. Medical image registration. *Physics in Medicine and Biology*, 46(3):R1–R45, 2001.

[42] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[43] H. P. Hofstee. Power-constrained microprocessor design. In *Proc. ICCD*, pages 14–16, 2002.

[44] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, 2005.

[45] M. Holden, J. Schnabel, and D. Hill. Quantification of small cerebral ventricular volume changes in treated growth hormone patients using nonrigid registration. *IEEE Trans. Med. Imag.*, 21(10):1292–1301, 2002.

[46] W.-m. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi, A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *Proc. DAC*, pages 754–759, 2007.

[47] F. Ino, K. Ooyama, A. Takeuchi, and K. Hagihara. Design and implementation of parallel nonrigid image registration using off-the-shelf supercomputers. In *Proc. MICCAI*, pages 327–334, 2003.

[48] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[49] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[50] A. Khamene, R. Chisu, W. Wein, N. Navab, and F. Sauer. A novel projection based approach for medical image registration. In *Proc. WBIR*, volume 4057 of *Lecture Notes in Computer Science*, pages 247–256. Springer, 2006.

[51] E. Kilgariff and R. Fernando. *GPU Gems 2*, chapter The GeForce 6 Series GPU Architecture, pages 471–492. Addison-Wesley Professional, 2005.

[52] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[53] A. Köhn, J. Drexl, F. Ritter, M. König, and H. O. Peitgen. *Bildverarbeitung für die Medizin 2006*, chapter GPU Accelerated Image Registration in Two and Three Dimensions, pages 261–265. Springer, 2006.

[54] V. Kumar and A. Gupta. Analysis of scalability of parallel algorithms and architectures: a survey. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 396–405, 1991.

[55] V. Kumar and V. N. Rao. Parallel depth first search. part ii. analysis. *Int. J. Parallel Program.*, 16(6):501–519, 1987.

[56] J. Kybic, P. Thévenaz, A. Nirkko, and M. Unser. Unwarping of unidirectionally distorted EPI images. *IEEE Trans. Med. Imag.*, 19(2):80–93, 2000.

[57] J. Kybic and M. Unser. Fast parametric elastic image registration. *IEEE Trans. Image Process.*, 12(11):1427–1442, 2003.

[58] M. Ledesma-Carbayo, J. Kybic, M. Desco, A. Santos, M. Sühling, P. Hunziker, and M. Unser. Spatio-Temporal Nonrigid Registration for Ultrasound Cardiac Motion Estimation. *IEEE Trans. Med. Imag.*, 24(9):1113–1126, 2005.

[59] B. Li, A. A. Young, and B. R. Cowan. GPU accelerated non-rigid registration for the evaluation of cardiac function. In *Proc. MICCAI (2)*, pages 880–887, 2008.

[60] B. Likar and F. Pernuš. A hierarchical approach to elastic registration based on mutual information. *Image and Vision Computing*, 19:33–44, 2001.

[61] J. A. Little, D. L. G. Hill, and D. J. Hawkes. Deformations incorporating rigid structures. *Comput. Vis. Image Underst.*, 66(2):223–232, 1997.

[62] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. K. Panda, and P. Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, 2004.

[63] D. Luebke and G. Humphreys. How GPUs work. *Computer*, 40(2):96–100, 2007.

[64] F. Maes, A. Collignong, D. Vandermeulen, G. Marchal, and P. Suetens. Multimodality image registration by maximization of mutual information. *IEEE Trans. Med. Imag.*, 16(2):187–198, 1997.

[65] J. Maintz and M. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.

[66] J. B. A. Maintz, E. H. W. Meijering, and M. A. Viergever. General multimodal elastic registration based on mutual information. In *Image Processing*, pages 144–154. SPIE Press, 1998.

[67] D. Mattes, D. R. Haynor, H. Vesselle, T. K. Lewellen, and W. Eubank. PET-CT image registration in the chest using free-form deformations. *IEEE Trans. Med. Imag.*, 22(1):120–128, 2003.

[68] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[69] S. A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 162–167, 2004.

[70] K. McLeish, D. Hill, D. Atkinson, J. Blackall, and R. Razavi. A study of the motion and deformation of the heart due to respiration. *IEEE Trans. Med. Imag.*, 21(9):1142–1150, September 2002.

[71] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top500 supercomputing sites. `http://www.top500.org/`.

[72] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant. Fast deformable registration on the GPU: A CUDA implementation of demons. In *ICCSA '08: Proceedings of the 2008 International Conference on Computational Sciences and Its Applications*, pages 223–233, 2008.

[73] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Amora, and S. Kesavarapu. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. Res. Dev.*, 51(5):573–582, 2007.

[74] NVIDIA. *NVIDIA CUDA Programming Guide 2.0.* 2008.

[75] M. Ohara, H. Yeo, F. Savino, G. Iyengar, L. Gong, H. Inoue, H. Komatsu, V. Sheinin, S. Daijavad, and B. Erickson. Real-time mutual-information-based linear registration on the Cell Broadband Engine processor. In *Proc. ISBI*, pages 33–36, 2007.

[76] S. Ourselin, A. Roche, S. Prima, and N. Ayache. Block matching: A general framework to improve robustness of rigid registration of medical images. In *Proc. MICCAI*, pages 557–566, 2000.

[77] J. Owens. *GPU Gems 2*, chapter Streaming Architectures and Technology Trends, pages 457–470. Addison-Wesley Professional, 2005.

[78] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[79] S. Patel and W.-m. W. Hwu. Accelerator architectures. *IEEE Micro*, 28(4):4–12, 2008.

[80] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[81] J. P. W. Pluim, J. B. A. Maintz, and M. A. Viergever. Mutual information based registration of medical images: A survey. *IEEE Trans. Med. Imaging*, 22(8):986–1004, 2003.

[82] V. Podlozhnyuk. Histogram calculation in CUDA. Technical report, NVIDIA, 2007.

[83] F. J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only). In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, page 2, 1999.

[84] T. Rohlfing, C. R. Maurer, Jr., W. G. O'Dell, and J. Zhong. Modeling liver motion and deformation during the respiratory cycle using intensity-based free-form registration of gated MR images. *Medical Physics*, 31(3):427–432, 2004.

[85] T. Rohlfing and C. R. Maurer Jr. Nonrigid image registration in shared-memory multiprocessor environments with application to

brains, breasts, and bees. *IEEE Trans. Inf. Technol. Biomed.*, 7(1):16–25, 2003.

[86] D. Rueckert, L. Sonoda, C. Hayes, D. Hill, M. Leach, and D. Hawkes. Nonrigid registration using free-form deformations: Application to breast MR images. *IEEE Trans. Med. Imag.*, 18(8):712–721, 1999.

[87] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[88] M. Sermesant, O. Clatz, Z. Li, S. Lanteri, H. Delingette, and N. Ayache. A parallel implementation of non-rigid registration using a volumetric biomechanical model. In *Proc. WBIR'03*, volume 2717 of *Lecture Notes in Computer Science*, pages 398–407. Springer-Verlag, 2003.

[89] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert. GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. *J. Phys. Med. Biol.*, 52(19):5771–5783, 2007.

[90] G. Soza, M. Bauer, P. Hastreiter, C. Nimsky, and G. Greiner. Non-rigid registration with use of hardware-based 3D Bézier functions. In *Proc. MICCAI*, pages 549–556, 2002.

[91] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. N. Strenski, and P. G. Emma. Optimizing pipelines for power and performance. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 333–344, 2002.

[92] R. Stefanescu, X. Pennec, and N. Ayache. Grid-enabled non-rigid registration of medical images. *Parallel Processing Letters*, 14(2):197–216, 2004.

[93] R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow — a streaming implementation in DX9 graphics hardware. *Computing*, 73(4):373–389, 2004.

[94] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[95] R. Szeliski and J. Coughlan. Spline-based image registration. *Int. J. Comput. Vision*, 22(3):199–218, 1997.

[96] P. Thévenaz and M. Unser. Spline pyramids for inter-modal image registration using mutual information. In *Proc. SPIE*, volume 3169, pages 236–247, 1997.

[97] P. Thévenaz and M. Unser. Optimization of mutual information for multiresolution image registration. *IEEE Trans. Image Process.*, 9(12):2083–2099, 2000.

[98] J.-P. Thirion. Non-rigid matching using demons. In *Proc. CVPR*, pages 245–251, 1996.

[99] J.-P. Thirion. Image matching as a diffusion process: An analogy with Maxwell's demons. *Medical Image Analysis*, 2(3):243–260, 1998.

[100] M. Unser, A. Aldroubi, and M. Eden. B-Spline signal processing: Part I—Theory. *IEEE Trans. Signal Process.*, 41(2):821–833, February 1993.

[101] M. Unser, A. Aldroubi, and M. Eden. B-Spline signal processing: Part II—Efficient design and applications. *IEEE Trans. Signal Process.*, 41(2):834–848, 1993.

[102] C. Vetter, C. Guetter, C. Xu, and R. Westermann. Non-rigid multimodal registration on the GPU. In *Proc. SPIE*, volume 6512, 2007.

[103] P. Viola and W. M. Wells III. Alignment by maximization of mutual information. *Int. J. Comput. Vision*, 24(2):137–154, 1997.

[104] S. Warfield, F. Talos, A. Tei, A. Bharatha, A. Nabavi, M. Ferrant, P. Black, F. Jolesz, and R. Kikinis. Real-time registration of volumetric brain mri by biomechanical simulation of deformation during image guided neurosurgery. *Computing and Visualization in Science*, 5(1):3–11, 2002.

[105] D. H. Woo and H.-H. S. Lee. Extending Amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[106] W. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. Technical report, University of Virginia, 1994.

[107] XtremeData. FPGA acceleration in HPC: A case study in financial analytics. Technical report, 2006.

[108] Z. Xu and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP2. *IEEE Parallel Distrib. Technol.*, 4(1):9–23, 1996.

[109] B. Zitová and J. Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, 2003.

Curriculum Vitae

Personal Data

Name	Jonathan Rohrer
Date of birth	July 16, 1980
Citizenship	Swiss

Jonathan Rohrer is member of the Hardware Acceleration Technologies group in the Systems Department at the IBM Zurich Research Laboratory. He received an M.S. degree in Electrical Engineering and Information Technology from the Swiss Federal Institute of Technology (ETH) Zurich in 2006. He subsequently joined IBM, where he works on programmable hardware accelerator units for different applications such as regular expression matching and protocol processing. Related to the PhD studies he is pursuing at the Computer Vision Laboratory at ETH Zurich, he works on nonrigid image registration and its acceleration. He is author or co-author of various papers and inventor or co-inventor of four filed patent applications.

Education and Degrees

2006 – 2009	PhD student Swiss Federal Institute of Technology (ETH) PreDoc IBM Zurich Research Laboratory
2000 – 2006	Master of Science in Electrical Engineering and Information Technology Swiss Federal Institute of Technology (ETH)
1993 – 2000	Matura Type C (Mathematics and Sciences) Kantonsschule Zug

Journals, Conferences and Workshops

J. Rohrer, J. van Lunteren, K. Atasu, and C. Hagleitner. Memory-Efficient Distribution of Regular Expressions for Fast Deep Packet Inspection. In *Proc. CODES+ISSS*, pages 147–154, 2009.

J. Rohrer and L. Gong. Accelerating 3D nonrigid registration using the Cell/B.E. processor. *IBM J. Res. Dev.*, 53(5):12:1–12:10, 2009.

J. van Lunteren, J. Rohrer, K. Atasu, and C. Hagleitner. Regular expression acceleration at multiple tens of Gb/s. In *1st Workshop on Accelerators for High-performance Architectures in conjunction with ICS*, 2009.

J. Rohrer, L. Gong, and G. Székely. Parallel mutual information based 3D non-rigid registration on a multi-core platform. In *HP-MICCAI workshop in conjunction with MICCAI*, 2008.

J. Rohrer and L. Gong. Focused atlas-based image registration for recognition. In *Proc. ICIP*, pages 1808–1811, 2008.

L. Gong, J. Rohrer, G. Iyengar, B. Butler, and A. Lumsden. Anatomical object recognition and labeling by atlas-based focused non-rigid registration and region-growing. In *Proc. ICALIP*, pages 1354–1358, 2008.

L. Gong, J. Rohrer, G. Iyengar, B. Butler, A. Lumsden, and P. Sovelius. Automatic labeling of blood vessels in CT abdominal images by progressive atlas-based registration and region-growing. *Int J CARS*, 3:S386–S387, 2008.

J. Rohrer and L. Gong. Accelerating mutual information based 3D non-rigid registration using the Cell/B.E. processor. In *Workshop on Cell Systems and Applications in conjunction with ISCA*, 2008.

J. Rohrer. Elastic Image Registration. In *Cell/B.E. on CNGrid Symposium*, Beijing, 2007.