GPGPU for Accelerated GRAPPA Autocalibration in Magnetic Resonance Imaging

Studienarbeit im Fach Informatik

vorgelegt von

Matthias Schneider

geb. am 17. Oktober 1984 in Aachen

angefertigt am

Institut für Informatik

Lehrstuhl für Graphische Datenverarbeitung

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Quirin Meyer, Frank Enders

Betreuender Hochschullehrer: Prof. Dr. Günther Greiner

Beginn der Arbeit: 01. September 2007

Abgabe der Arbeit: 01. April 2008

Contents

1 Introduction 1

1.1 Related Work . . . 2

1.2 Contributions . . . 2

1.3 Outline . . . 3

2 Magnetic Resonance Imaging 5

2.1 Overview and History . . . 5

2.2 MR Physics . . . 6

2.2.1 Fundamentals of Spin Physics . . . 6

2.2.2 Precession . . . 8

2.2.3 RF Pulses and Resonance Condition . . . 9

2.2.4 MR Signal . . . 9

2.2.5 Spin Relaxation . . . 11

2.2.6 Spin Echo . . . 12

2.3 Image Reconstruction . . . 14

2.3.1 Spacial Allocation and Slices . . . 14

2.3.2 Frequency and Phase Encoding . . . 15

2.3.3 Pulse Sequence . . . 17

2.4 Contrast Techniques . . . 18

2.5 Parallel Acquisition Techniques . . . 20

2.5.1 Motivation . . . 20

2.5.2 Reconstruction in Image Domain . . . 22
2.5.3 Reconstruction in k-space . . . 24
2.6 The GRAPPA Algorithm in Detail . . . 25
2.6.1 Reconstruction of k-space . . . 25
2.6.2 Autocalibration . . . 27
2.6.3 Further Development . . . 30
2.7 Imaging Hardware . . . 31
3 General-Purpose Computing on GPUs 33
3.1 Introduction . . . 33
3.1.1 Why GPUs for General-Purpose Computing? . . . 33
3.1.2 Limitations and Requirements . . . 35
3.2 Programmability of GPUs . . . 35
3.2.1 Graphics Pipeline . . . 35
3.2.2 High-Level Languages . . . 36
3.3 Compute Unified Device Architecture . . . 37
3.3.1 Overview . . . 38
3.3.2 Programming Model . . . 38
3.3.3 Execution Model . . . 39
3.3.4 Memory Model . . . 40
3.3.5 Hardware . . . 42
3.3.6 Designing Parallel Algorithms . . . 42
3.3.7 Performance Aspects . . . 43
3.4 GPGPU in Practice . . . 46
4 GPGPU for GRAPPA Autocalibration 47
4.1 Autocalibration Algorithm in Practice . . . 47
4.1.1 Basic Approach . . . 48

4.1.2 Improved Approach . . . 50
4.1.3 Computational Costs and Complexity . . . 54
4.2 Matrix Multiplication . . . 57
4.2.1 Basic Approach . . . 57
4.2.2 Improved Kernels . . . 60
4.2.3 CUBLAS . . . 66
4.2.4 Special case . . . 68
4.3 Initialization and Normalization . . . 70
4.3.1 Initialization . . . 70
4.3.2 Normalization . . . 71
4.4 Entire Autocalibration Stage . . . 74
5 Results 77
5.1 Matrix Multiplication . . . 78
5.2 Matrix Inversion . . . 82
5.3 Entire Autocalibration Stage . . . 85
5.3.1 Standalone Autocalibration . . . 85
5.3.2 Integrated Autocalibration . . . 90
5.3.3 Computational Error . . . 92
6 Conclusion 95
6.1 Summary . . . 95
6.2 Future Work . . . 96
6.3 Final Remarks on CUDA . . . 97
A Notation and Preliminaries 103
A.1 Matrix Structure . . . 103
A.2 Matrices in Memory . . . 103

A.3 Segmented Matrices 104
A.4 Implicit Variables 105

List of Acronyms 107

List of Symbols 109

Bibliography 111

List of Figures

2.1 Magnetic Effect of Spinning Proton [29] 7
2.2 Voxel Magnetization [29] 7
2.3 Spin Precession [29] 8
2.4 Influence of 90 Degree RF Pulse on Spin Precession [29] 10
2.5 Course of MR Signal [29] 11
2.6 Tissue Specific T_1 and T_2 [29] 12
2.7 Spin Echo [29] 13
2.8 Gradients for Slice Selection [29] 15
2.9 Frequency Encoding and Decoding [29] 16
2.10 Pulse Diagram [29] 18
2.11 Contrast Weighting [29] 19
2.12 Basic pMRI Concepts [9] 22
2.13 SENSE Reconstruction Scheme [3] 23
2.14 GRAPPA at a Glance 26
2.15 GRAPPA Reconstruction Scheme 28

3.1 Trend of Peak Performance [78] 34
3.2 Graphics Pipeline 36
3.3 CUDA Memory Model 41

4.1 Matrix Structure in Basic GRAPPA Autocalibration 49
4.2 Matrix Structure in Improved GRAPPA Autocalibration 52

4.3 Basic Scheme for Matrix Multiplication 58
4.4 Adjusted Scheme for Matrix Multiplication 60
4.5 Optimized Scheme for Matrix Multiplication 67

5.1 Comparison of `mmul[1-4]` and `cublasCgemm()` 78
5.2 Comparison of `mmul[4,5]`, `cublasCgemm()`, and Intel MKL 80
5.3 Comparison of `mmul[4,5]`, `cublasCgemm()`, and Intel MKL 81
5.4 Results for `findWs[GPU]` Depending on N_{col} 86
5.5 Breakdown of Overall `findWs[GPU]` Execution Time 87
5.6 Effective Memory Throughput During Initialization of A and B 88
5.7 Results for `findWs[GPU]` Depending on N_c 89
5.8 Comparison of GPU to CPU for Practical Test Cases 91
5.9 Outlook on Future Number of Channels 92

List of Tables

2.1 Parameter Choice for Gradient Echo Sequences [29] 20

3.1 Comparison of Recent GPUs and CPUs 34

3.2 GPU Shading Language Overview 36

3.3 Overview of GPGPU High-Level Languages 37

4.1 Matrix Dimensions in GRAPPA Autocalibration 56

4.2 Computational Complexity of GRAPPA Autocalibration 56

4.3 Access Pattern with Bank Conflicts 63

4.4 Conflict-Free Access Pattern 63

4.5 Dispatching for Initialization of A and B 71

5.1 Default Values of GRAPPA Parameters 86

5.2 GPU Memory Requirements of `findWsGPU` 90

A.1 Accessing Matrices 104

A.2 Index and Dimension Tags for CUDA Threads 105

List of Algorithms and Listings

3.1 Launching a CUDA Kernel . . . 40
4.1 Steps of GRAPPA Autocalibration Stage . . . 47
4.2 GRAPPA Autocalibration (findWs) . . . 50
4.3 Improved GRAPPA Autocalibration (findWsImproved) . . . 53
4.4 Basic Approach for Matrix Multiplication (mmul1) . . . 59
4.5 Improved Approach for Matrix Multiplication (mmul2) . . . 61
4.6 Improved Approach for Matrix Multiplication (mmul3) . . . 64
4.7 Improved Approach for Matrix Multiplication (mmul4) . . . 65
4.8 Improved Matrix Multiplication $Z = X \cdot X^H$ (mmul5) . . . 69
4.9 Matrix Transpose (mTrans) . . . 73
4.10 Matrix Normalization (mNorm) . . . 73
4.11 GRAPPA Autocalibration Pipeline Using GPGPU (findWsGPU) . . . 74
5.1 Matlab Implementation for Outer Product Cholesky Decomposition (cholDec) . . . 83

Chapter 1

Introduction

The success story of *Magnetic Resonance Imaging (MRI)* dates back to 1946 when the phenomenon of *Magnetic Resonance (MR)* was discovered. From that time on, MRI has emerged as a powerful modality in medical imaging.[1] Quite a lot of research has been carried out to further and further improve hardware performance and image quality. MRI in its infancy had to overcome difficulties particularly concerned with the generation of strong, homogeneous, and static magnetic fields and gradients. Meanwhile, the tide has turned and algorithms for image reconstruction have reached a bottleneck for clinical real-time applications in particular. Current MR systems use ingenious parallel image acquisition techniques such as *Generalized Autocalibrating Partially Parallel Acquisitions (GRAPPA)* involving computationally intensive reconstruction algorithms. As a result, the acquisition time is reduced at the expense of reconstruction time limited by the computational power provided by CPUs.

One way to improve algorithmic performance, that has become increasingly popular in recent years, is to take advantage of the computational power of Graphics Processing Units (GPUs) for general-purpose computations. The increase of the theoretical GPU peak performance has exceeded by far the augmentation of CPU performance during the last years. This trend is doubtlessly pushed forward by the game industry in particular because of more and more detailed and realistic real-time graphics in videogames. Modern graphics hardware is equipped with 16 to 64 multiprocessors simultaneously executing single instructions on multiple data and provides "enough" on-board memory

[1] Approximately 10,000 MRI units worldwide and 75 million MRI scans performed per year [32].

(512 MB up to 1.5 GB) to store large data sets. Since 2006, the general programmability of GPUs has largely increased, which makes them also suitable for science and engineering applications.

1.1 Related Work

Medical imaging was indeed one of the first general-purpose applications for GPUs. Research has primarily focused on accelerating reconstruction algorithms for *Computed Tomography (CT)*, especially the *Filtered Backprojection* and the *Fast Fourier Transform (FFT)* [13,50,51,75]. Besides GPUs, there are also some promising approaches based on other architectures such as *Field Programmable Gate Arrays (FPGAs)* [81] and the *Cell Broadband Engine* [74].
Algorithms for Filtered Backprojection and FFT also play a crucial role in MRI reconstruction and have been implemented for GPUs resulting in speedups between two and nine [79, 76]. Recently, an advanced least-squares reconstruction algorithm operating on non-Cartesian scan data has been accelerated by state-of-the-art graphics hardware. Experimental optimization techniques achieve speedup factors up to 120, which shortens the reconstruction time from six hours to three minutes [78].

1.2 Contributions

This thesis describes how to leverage the computational power of recent NVIDIA GPUs supporting the *Compute Unified Device Architecture (CUDA)* for the GRAPPA autocallibration stage that takes 25% of the total reconstruction time on CPUs. The computationally most intensive and hence most time-consuming part of the reconstruction algorithm is to solve large, overdetermined linear equation systems. The involved complex-valued matrix multiplications usually make up more than 80% on CPU. We present different CUDA-based implementations to accelerate this task that can be parallelized to map well to the underlying parallel hardware architecture. Our optimized kernel for matrix multiplication on GPUs performs eleven times faster than the highly optimized Intel *Math Kernel Library (MKL)* and even up to 17 times faster for special cases.
For current configurations, the GPU-accelerated parts of the GRAPPA Autocalibration stage perform 30 to 45 times faster compared to the the CPU-based implementation that is in use in MR systems of Siemens Medical Solutions for some years now. The reconstruction time is considerably reduced

so that the application of future coil arrays with up to 128 channels is achievable. All in all, the GPU-powered GRAPPA autocalibration now only makes up two to three percent of the overall reconstruction time.

1.3 Outline

This thesis is structured as follows:

Chapter 2 provides an overview of *Magnetic Resonance Imaging* including different imaging and reconstruction techniques, particularly GRAPPA.

Chapter 3 describes the concept of GPGPU with focus on NVIDIA's CUDA.

Chapter 4 demonstrates how GPGPU can be used for the GRAPPA autocalibration stage including several optimization techniques.

Chapter 5 evaluates different benchmark tests and compares our improved GPU-accelerated approach to present CPU-powered implementations.

Chapter 6 concludes the results of this thesis and gives an outlook to future work.

Please also note Appendix A for notation and preliminaries.

Chapter 2

Magnetic Resonance Imaging

2.1 Overview and History

Just as the well-known ultrasonic and X-ray examination, *Magnetic Resonance Imaging (MRI)* is a non-invasive imaging technique primarily used for medical purposes. It is based on the principles of *Nuclear Magnetic Resonance (NMR)*. In 1946, Felix Bloch [6, 7] and Edward Purcell [72] independently discovered the phenomenon of *Magnetic Resonance (MR)* and used it for chemical and physical molecular analysis. For their invention they were awarded the Nobel Prize in Physics 1952 [52].
It was Paul Lauterbur who suggested using NMR to discriminate two different mediums in 1973. Four years later, his proposal was finally refined by Peter Mansfield to a fast imaging technique. Both scientists were awarded the Nobel Price in Medicine 2003 [53].

MRI is an incredibly powerful modality and a widely-used diagnostic technique for medical purposes. Tissue structures and anatomic details can be displayed in a series of high quality slice images of any oblique plane through the human body, without the patient ever moving [18]. This is a major advantage of MRI compared to X-ray CT scanners that are, above all, limited to one plane (axial, coronal or sagital). The outcome of this is both a minimized examination time and precise, satisfactory diagnostics.
In contrast to radiotechnology, MRI works with non-ionizing radiation of much lower energy and achieves excellent soft tissue contrast at a higher resolution compared to ultrasonic imaging techniques [8, 29]. Due to its brilliant image quality and soft tissue resolution, MRI is used, among other

things, for comprehensive imaging of the heart, contrast-enhanced angiography, proton spectroscopy, diffusion and perfusion imaging, as well as in gastroenterology, orthopedics, and neurology (neuro-imaging) [29].

Medical examinations using an MR scanner proceed in several steps: First of all, the patient is positioned in the magnet of the device that exposes his body to a strong magnetic field. In the course of the examination the atomic nuclei of the patient's body are stimulated to emit high-frequency radiation as described in Chapter 2.2.1. This measurable signal is captured by an appropriate receiver and is finally processed within the image reconstruction stage resulting in a slice image of the body.

Understanding the entire functional principle of MRI in depth is no bed of roses, as two Nobel Prizes already suggest. Nevertheless, some fundamentals of nuclear physics are needed to get an idea of the basic principles of MRI.

2.2 MR Physics

MRI uses the magnetic properties of hydrogen nuclei for imaging. Hydrogen is the chemical element with the least complex nucleus consisting of one single proton with a positive electrical charge. As it turns out, this is the element with the strongest magnetic resonance signal an hence particularly suitable for MRI. Fortunately, hydrogen, as an elementary component of fat and water, is the most prevalent element in the human body.

2.2.1 Fundamentals of Spin Physics

Protons, electrons, and neutrons possess a quantum-mechanical characteristic referred to as *spin*. We can imagine this fundamental property as a never-ending rotation of a sphere, even if the proton actually doesn't spin itself. Moreover, the spin causes a nucleus to have a magnetic effect comparable to a tiny bar magnet with a magnetic north (N) and south (S) pole as shown in Figure 2.1. The spin is a unique, directional quantity of a nucleus. It can be characterized by a vector of certain length but variable direction (from S to N) corresponding to the rotation axis.

However, MRI does not measure the individual spin of a single nucleus but the effect of an entire collection (*ensemble*) of spins within a volume element (*voxel*) of the human body. According to the

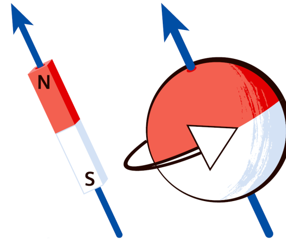


Figure 2.1: Magnetic Effect of Spinning Proton

Pauli Exclusion Principle [69], the overall effect, that is observable from the outside, results from superimposition, i.e. spatial addition of the single nuclei spin vectors.
In field-free space without an external magnetic field, the spin effects cancel out each other as they are randomly oriented within the ensemble. Consequently, it appears to be macroscopically non-magnetic.

Exposing the voxel to a static external magnetic field, however, a force acts on the spins that are oriented longitudinally to the magnetic field. In contrast to the comparable model of bar magnets, the spins behave differently. They align partially parallel (*spin up*) and partially anti-parallel (*spin down*) to the field. These two states, the protons can occupy, correspond to energy states separated by a quantum of energy. The spins distribute almost equally but with a small majority of *excess spins* (spin up) of lower energy. Macroscopically considered, this *state of equilibrium* leads to a weak magnetization (*M*) of the voxel as illustrated in Figure 2.2.

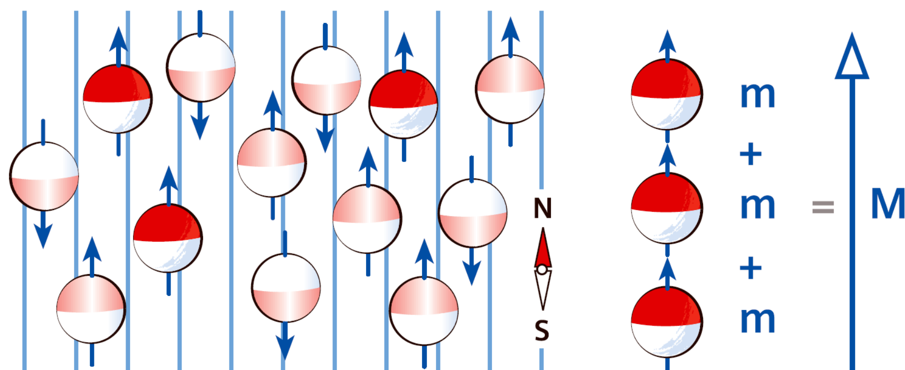


Figure 2.2: Voxel Magnetization

The number of excess spins rises with the strength of the external magnetic field, the proton density within the voxel, and also depends on the temperature. There are approximately six excess spins per one million protons for instance at body temperature and a field strength of one Tesla (T). Fortu-

nately, a cubic voxel of water with an edge length $s = 1$ mm contains approximately $6.7 \cdot 10^{19}$ hydrogen protons, which results in $4.0 \cdot 10^{14}$ (400 trillion!) excess spins. They all contribute to a macroscopic magnetization so that the quite small amount of excess spins is by no means a drawback.

Up to now, the spin model only explains the magnetization along the field lines. For further considerations, however, we necessarily have to refine this model.

2.2.2 Precession

Strictly speaking, the spins which are subject to a static magnetic field do not align exactly parallel or anti-parallel to the field lines as stated in the previous section, but they spin like a tilted top. The lower tip of the top stands still whereas the upper one continuously moves on a circular path. So this means, the spin vector moves in a conic shape around the direction of the external field. This type of spinning movement also known as *precession* is illustrated in Figure 2.3.

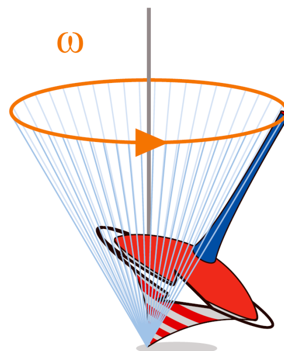


Figure 2.3: Spin Precession

The rotation speed of a precessing spin depends on the nature of the nucleus and increases with the strength of the applied magnetic field B. This characteristic frequency is also referred to as *Larmor frequency* ω with

$$\omega = \gamma B\,, \tag{2.1}$$

where γ is the *gyromagnetic ratio* of the nucleus ($\gamma_{^1H} = 42.58\,\mathrm{MHz/T}$ for hydrogen [32]).
For very strong magnetic fields as prevalent in MR systems, spins oscillate at radio frequency. The Larmor frequency of hydrogen for example reaches approximately 43 MHz at a magnetic field $B = 1.0$ T

and $\omega = 63\,\text{MHz}$ for $B = 1.5\,\text{T}$ respectively.

The excess spins, thus, all precess at the same frequency about the direction of the external magnetic field (vertical z-axis). Nevertheless, they precess out-of-phase with a randomly distributed phase difference in the *basic state*. In other words, the horizontal components transverse to the magnetic field, i.e. parallel to the xy-plane, cancel out each other across the entire ensemble and a constant magnetization exists only along the z-axis (see Figure 2.4a).

2.2.3 RF Pulses and Resonance Condition

We can deflect the energy equilibrium of the precessing spins in the basic state, though, by using a suitable electro magnetic wave. This radio frequency wave is also known as *RF pulse* and has to fulfill the so-called *resonance condition*. It states, to put it simple, that the RF pulse interferes with spins only if its oscillating frequency matches the spins' Larmor frequency.

The spin vectors tilt by a specific angle (*flip angle*) in the basic state depending on the duration and the energy of the interfering RF pulse. Thus, they can particularly be rotated by 90 degrees into the xy-plane or even flipped by 180 degrees into the opposite z-direction. The direction of the net magnetization of course changes accordingly. These two types of interfering RF pulses play an important role in MR imaging and allow to make the spins generate an MR signal.

2.2.4 MR Signal

As mentioned before, the original longitudinal magnetization in the direction of the z-axis is rotated into the xy-plane at the end of a 90 degree pulse (see Figure 2.4b). Since the z-components of the rotated spin vectors neutralize each other, the longitudinal magnetization is zero. But things look completely different after the pulse. The spins are then only exposed to the static external magnetic field and rotate about the z-axis. Nevertheless, there is a remarkable difference compared to the basic state: The transverse magnetization is still prevalent in the xy-plane as the spins are phase-coherent, now. In other words, the original longitudinal magnetization is flipped by 90 degrees acting like a rotating magnet in the xy-plane (see Figure 2.4c). According to the theory of RF induction, this rotation produces an electric voltage in a receiver coil. The course of this voltage over time is the *MR signal*. The stronger the transverse magnetization, the stronger the MR signal [29]. As shown in Figure 2.5,

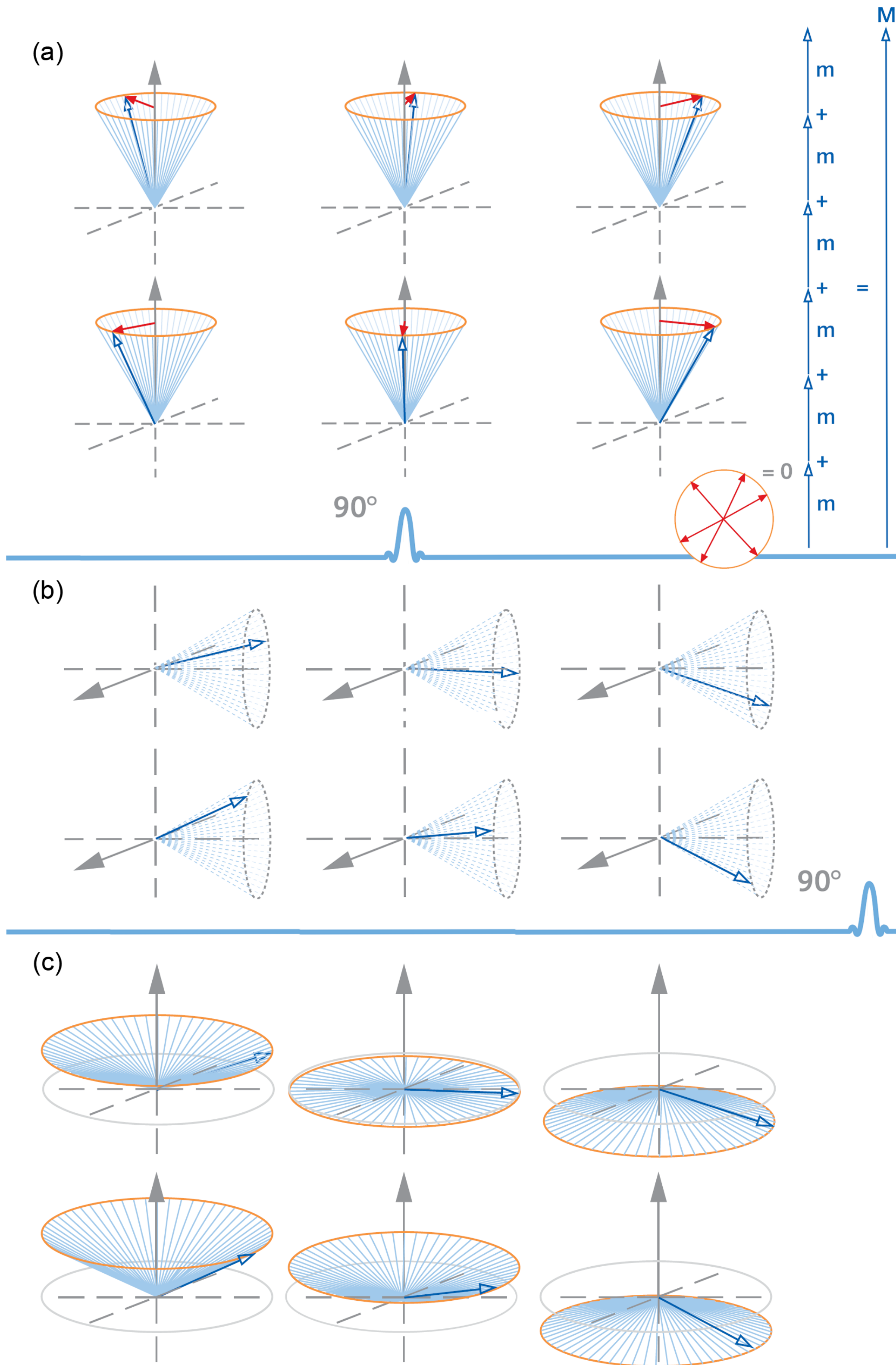


Figure 2.4: Spin precession before (a), at the end (b), and after (c) a 90 degree RF pulse.

the shape of the signal is similar to a damped oscillation. It decays quickly after the end of the RF pulse and is therefore also called the *Free Induction Decay (FID)*.

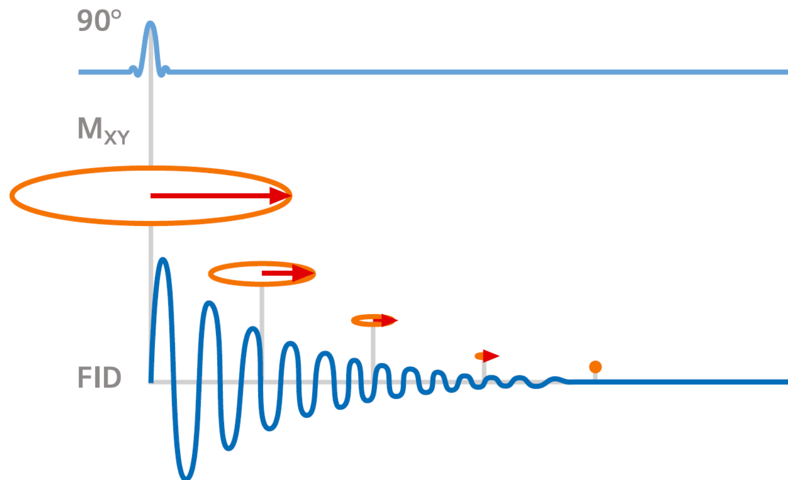
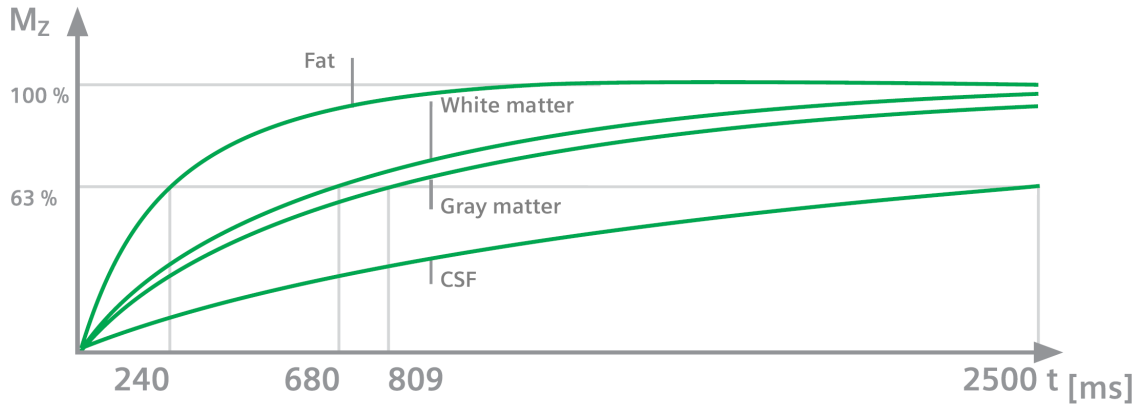


Figure 2.5: Course of MR Signal over Time

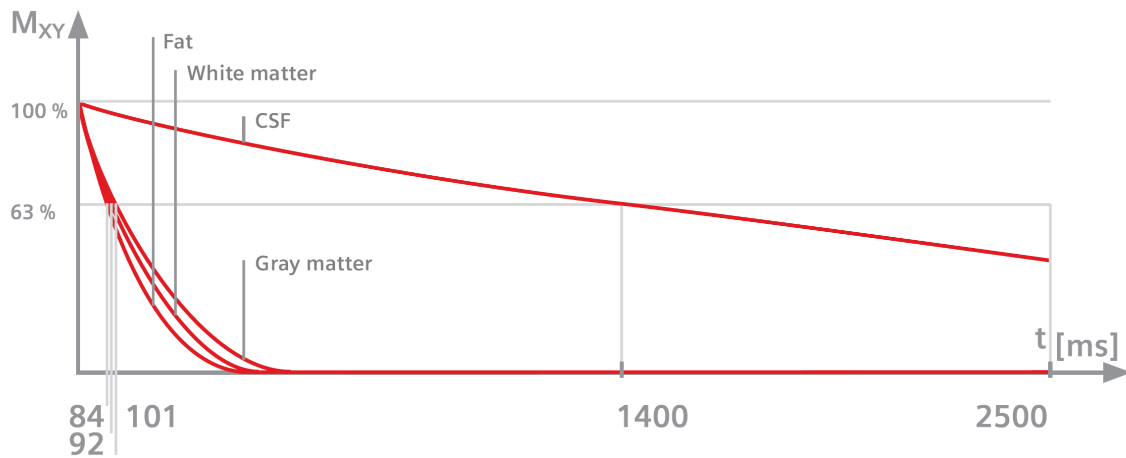
2.2.5 Spin Relaxation

The spins in the interfered state return to the original state of equilibrium with longitudinal but no transverse magnetization. This dynamic process of excited-state deactivation is called *relaxation*. It is the cause of the rather quickly decay of the MR signal. Relaxation itself is mainly caused by exiguous magnetic field fluctuations especially in the range of the Larmor frequency. This magnetic noise is generated by molecular motion.

The *transverse magnetization* M_{XY} dies down faster than the *longitudinal magnetization* M_Z regrows. Both processes follow an exponential function of time that is characterized by two time constants T_1 for M_{XY} and T_2 for M_Z, whereby $T_2 < T_1$. The time constants are defined such that the value of the exponential function evaluated at T is 63% of the final value which is all but reached after $5T$. For the sake of completeness, it should be mentioned that the T_1 process is also called *spin-lattice relaxation* and the T_2 process *spin-spin relaxation*. The difference of the two time constants mainly results from spin-spin interactions [29]. Fortunately, the time constants T_1 and T_2 are tissue-specific as illustrated in Figure 2.6. This is the reason why MR signals are well suited to generate high-contrast MR images distinguishing different tissue types.



(a) Recovery of Longitudinal Magnetization M_Z for $B = 1.0\,\mathrm{T}$ (T_1)



(b) Decay of Transversal Magnetization M_{XY} (T_2)

Figure 2.6: Tissue Specific T_1 and T_2

2.2.6 Spin Echo

Considering the MR signal, it turns out that it decays much faster than actually expected according to T_2. The observed effective time constant T_2^* results from static inhomogeneities of the external magnetic field. They are caused by the patient's body as well as inevitable technical inhomogeneities of the magnet. The rule of thumb is:

$$T_2^* < T_2 < T_1 \quad .$$

Unfortunately, it is not possible to measure the first peak of the MR signal due to the strong RF pulse that is still in effect at the beginning of the FID and would excessively affect the electromagnetic measuring. The solution to this problem uses a little trick. If we switch on a 180 degree pulse after run

time τ behind the 90 degree pulse, the so called *spin echo* arises and reaches its maximum after the so called *echo time* $T_E = 2\tau$. This effect is illustrated in Figure 2.7.

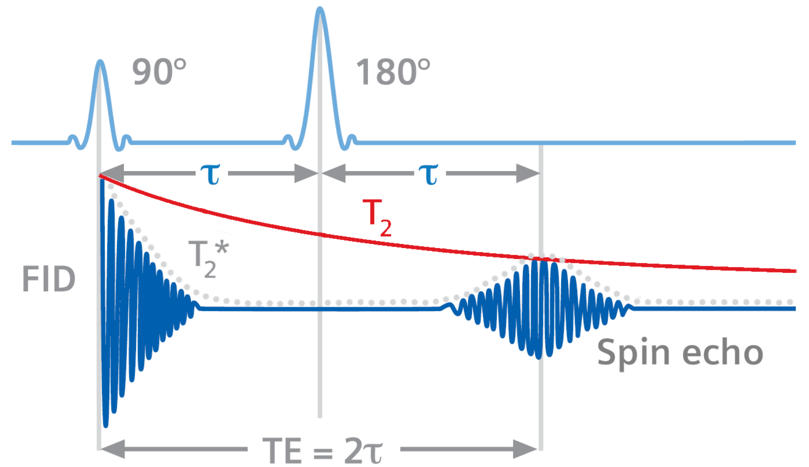


Figure 2.7: Spin Echo

In [29] this effect is explained intuitively using "The example of the runners" on a circular track. At the beginning of the race, all the runners toe the line. After the starter's gun, they start running at slightly different but constant speeds so that they run out of phase by-and-by. After a certain time τ, the runners about-face and run back without changing their speeds. After another run time τ, they all will simultaneously arrive at the starting line again.

The spins' situation is indeed comparable to this example. After the 90 degree pulse, the spins are phase-coherent. During the "race", they run completely out-of-phase. The about-face is accomplished by applying the 180 degree pulse after run time τ. Finally, the out-of-phase spins get back into phase. They generate a spin echo reaching its amplitude after 2τ. The condition of equal "speeds" before and after the about-face is also fulfilled since the inhomogeneities of the magnetic field remain the same over time at a specific location. In other words, the spins come across the same static magnetic field differences both before the 180 degree pulse and after it.

The generation of spin echos can even be iterated repeatedly by applying a *multi-echo sequence* consisting of consecutive 180 degree pulses whereby the amplitudes of the spin echos decrease with T_2.

2.3 Image Reconstruction

Different anatomical structures of the human body differ in their relaxation times leading to distinct MR signals. Therefore, the arising MR signals during an MR examination can be used to reconstruct medical images showing a high contrast for different tissues. For medical diagnostics, images of thin slices at specific positions through the patient's body are needed.

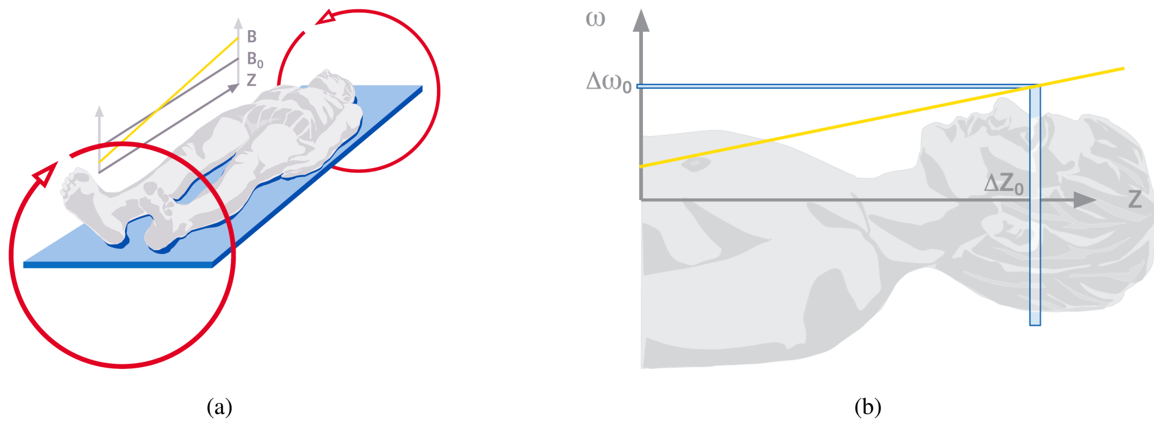
2.3.1 Spacial Allocation and Slices

The acquired MR signal results from the superimposition of all magnetic resonances in the examined body and does not allow for *spatial allocation* so far. MR scanners generate a homogeneous static magnetic filed typically by using large coils. Exposed to this field, all protons of the patient's body precess at a specific Larmor frequency ω_0 that depends on the strength of the magnetic field according to Equation 2.1. The described pulse technique causes each proton to generate an equal echo that contributes to a single MR signal which is measurable from the outside yet with completely lost spatial resolution. However, there is a simple answer to this problem: *gradients*.

If we spatially vary the static magnetic field B_0 by switching on a gradient in the direction of the z-axis, the Larmor frequency of the protons changes accordingly. The gradient is usually generated by two inversely phased coils roughly sketched in Figure 2.8(a). As a result, the strength of the magnetic field as well as the Larmor frequencies linearly increase along the gradient's direction. An RF pulse of frequency ω_0 now excites only spins at position z_0 corresponding to the Larmor frequency ω_0. In this way, a spatial region (*slice*) of resonating spins is selected. The associated gradient is known as *slice-selection gradient* (G_S).

Moreover, a certain *slice thickness* is needed for a signal of sufficient strength and satisfactory quality. This task is performed for instance by an RF pulse possessing a certain frequency bandwidth $\Delta\omega_0$ instead of a single frequency ω_0. This $\Delta\omega_0$ pulse[1] selects a slice of the corresponding thickness Δz_0, which is shown in Figure 2.8(b). Gradients even allow for selecting slice planes not only parallel to the xy-plane, but at arbitrary and oblique positions. For this purpose, several gradients along the three spatial axes are combined appropriately [29].

[1] Another way to deal with the problem is to use gradients of different slopes [29].



(a) (b)

Figure 2.8: Gradients for Slice Selection

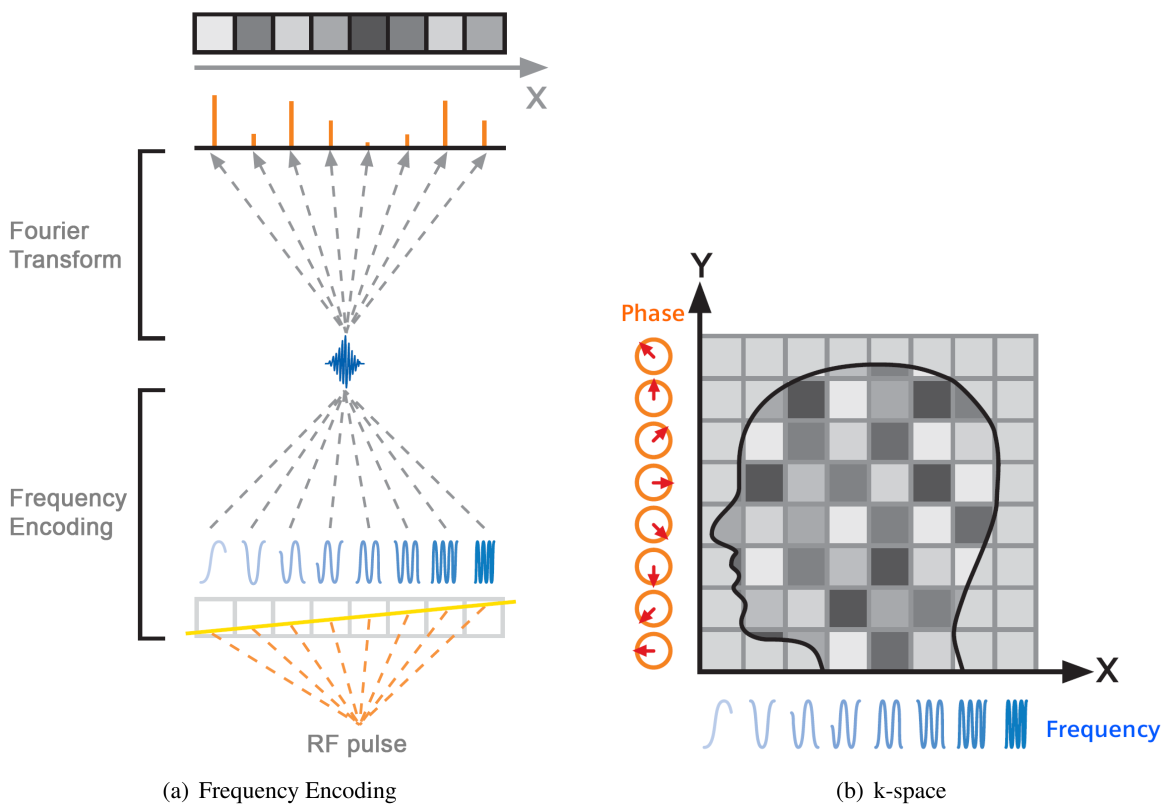
This gradient-based method for slice selection reduces the dimension of the spatial problem by one but we still have to deal with the remaining two-dimensional problem of assigning a gray value to each pixel of the MR image.

2.3.2 Frequency and Phase Encoding

An MR image consists of many picture elements (*pixels*), e.g. 256×256. Each pixel of the so called *image matrix* corresponds to a voxel of the selected slice. Accordingly, the pixels' gray values can be computed from the echo signal of the proper voxel. In order to make this calculation possible, we need further spatial allocation of the MR signal generated by a single slice. This problem can be solved once more by using the all-purpose tool within MRI: gradients.

The selected slice is assumed to be perpendicular to the z-axis for the moment. A gradient that is applied in the direction of the x-axis causes the spins to process at increasing oscillation frequencies along the x-direction (*frequency encoding*). In other words, the so called *frequency-encoding gradient* (G_F) leads to an MR signal consisting of as many different frequencies as there are pixels in a single row of the image matrix (256 in this case). If we apply the *Fourier Transform* [15], this composite signal is divided into its components of individual frequencies and amplitudes. Based on this decomposition of the MR signal, it is possible to assign proper gray values to the pixels of the rows within the image matrix. This frequency encoding and decoding process is illustrated in Figure 2.9(a).

Finally, there is left only the differentiation of the single rows in order to accurately localize an individual voxel within the selected slice. For this purpose, a *phase-encoding gradient* (G_P) is briefly switched on in the *y*-direction during the time between the RF pulse and the echo. As a result, the spins precess at different speeds until the gradient is switched off again and show different phase shifts directly proportional to their locations along the *y*-axis [29]. The Fourier Transform is able to filter out these phase shifts. All in all, the phase encoding step has to be repeated with different phase encodings for every image row. The resulting MR signals are stored in the rows of a *raw data matrix* shown in Figure 2.9(b). This matrix is also referred to as *k-space* and can finally be Fourier transformed to the final grayscale MR image. Due to the different types of encoding, the *x*-direction is also denoted as *Read-Out Direction (RO)* and the *y*-direction as *Phase-Encoding Direction (PE)*.



(a) Frequency Encoding (b) k-space

Figure 2.9: Frequency Encoding and Decoding

2.3.3 Pulse Sequence

In order to obtain an MR image, the different pulses and gradients are put together to a *pulse sequence*. Despite the enormous variety of sequences, there are in fact only two basic models: *Spin Echo (SE)* and *Gradient Echo (GE)* sequences.
Sequences within these two different classes slightly differ in terms of contrast technique (see Chapter 2.4) and imaging speed. The following example of a very simple pulse sequence in Diagram 2.10 is used to explain the main principle of pulse sequences:

Slice selection

The slice-Selection gradient G_S is in effect during both the 90 degree and the 180 degree pulse to delimit the effect of the HF pulses to a specific spatial region. As the spins are dephased by the gradient, another *rephasing gradient* is switched on (short trough after the 90 degree pulse) to rephase the spins along the slice thickness again.

Frequency encoding

The frequency-encoding gradient G_F is switched on during the readout of the spin echo. Strictly speaking, another rephasing gradient has to be included to compensate for the dephasing effect of G_F. This additional gradient is omitted in Figure 2.10 to keep things simple.

Phase encoding

The phase encoding gradient G_P is used between slice selection and spin echo. The phase encodings differ for every row in k-space determining the number of repetitions of the sequence.

The whole pulse sequence is continuously repeated after *repetition time* (T_R) according to the number of raw data rows in k-space. As $T_E \ll T_R$, it is even possible to excite several slices consecutively within repetition time (*multislice sequence*).

Other spin echo sequences use more elaborate pulse sequences that allow for accelerated MR data acquisition. The simple spin echo sequence for example can be extended by additional 180 degree pulses to obtain an entire series of echo signals within T_E (*echo train*). This *Fast Spin Echo Sequence (FSE)*, also known as *Turbo Spin Echo Sequence (TSE)*, significantly shortens the number of repetitions and hence the acquisition time (e.g. *turbo factor* between 7 and 15).

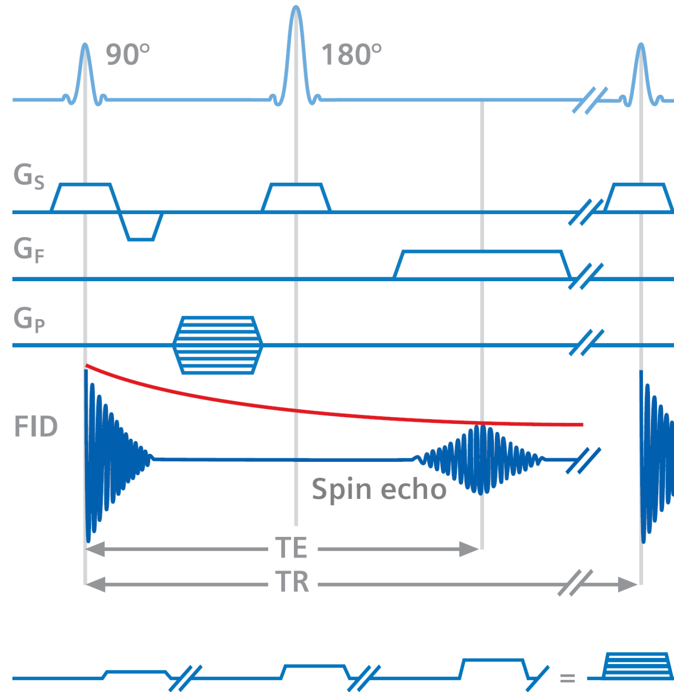


Figure 2.10: Pulse Diagram

An even faster acquisition method is the *Echo-Planar Imaging (EPI)*. A single 90 degree pulse is used to acquire an entire image. Regrettably, this technique is limited to “small” matrices (128×128) [29]. By contrast, gradient echo sequences start with an RF pulse of a smaller flip angle α and T_R and T_E are chosen much smaller compared to *SE* sequences.

Further information and detailed explanations concerning the different types of pulse sequences can be found for instance in [46] and [49].

2.4 Contrast Techniques

As described in Chapter 2.2.5, there are three properties of human tissue that allow for image contrast using spin echo sequences: *longitudinal relaxation time* T_1, *transversal relaxation time* T_2 and *proton density* ρ_{PD} *(PD)*, i.e. the number of hydrogen protons per volume unit [29].

They all contribute to the intensity S of an MR signal that is acquired by a pulse sequence with

repetition time T_R and echo time T_E according to the following equation [49]:

$$S \sim \rho\, e^{-T_E/T_2}\left(1 - e^{-T_R/T_1}\right)\ , \quad \text{where } T_E \ll T_1,\ T_2 \ll T_R\ .$$

For a suitable choice of T_R and T_E the signal intensity is primarily determined by one of the three tissue parameters ρ, T_1, and T_2. Accordingly, there are three different types of MR images: T_1-*weighted*, T_2-*weighted*, and *PD-weighted* images. Figure 2.11 roughly sketches the relationship between the choice of T_R and T_E and the weighting of the resulting image.

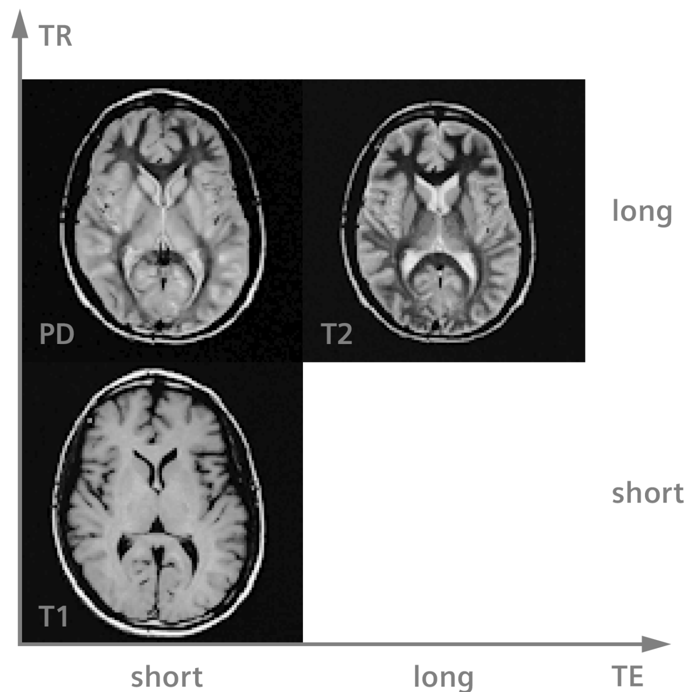


Figure 2.11: Contrast weighting for different combinations of T_E and T_R.

For gradient echo sequences, however, the type of image contrast is determined by both T_E and the flip angle α as shown in Table 2.1. A detailed discussion of the depicted relationship can be found in [46] and [29].

PD-weighted as well as T_1-weighted images are usually used to visualize anatomical structures as they clearly show the boundaries between different tissue types, whereas T_2-weighted images especially highlight pathological tissue such as tumors [49]. Moreover, there are various techniques

	T_R	T_E	α
T_1	short (40 – 150 ms)	short (5 – 10 ms)	large (40° – 80°)
T_2^*	long (500 ms)	long (18 – 40 ms)	small (5° – 10°)
PD	long (500 ms)	short (5 – 10 ms)	small (5° – 20°)

Table 2.1: Parameter Choice for Gradient Echo Sequences

in order to enhance the contrast of specific tissue. Therefore, ingenious pulse sequences are used to suitably combine the image contrast effects of the tissue parameters [46].

2.5 Parallel Acquisition Techniques

2.5.1 Motivation

Apart from image contrast, the acquisition time is one of the most important considerations in clinical applications not only because of the matter of efficiency and patient throughput but also for practical reasons. Simple spin echo sequences take several minutes to acquire clinically feasible images, which is much too long for a multitude of medical applications. Examinations of the breast, abdomen, or the pelvis particularly demand a short acquisition time as there is inevitable motion of the organs in these areas, e.g. beating of the heart, respiratory movement, or gut motility. In the late 1980s, the invention of fast imaging sequences, such as *Fast Low-Angle Shot (FLASH)*, EPI, and TSE met these requirements [46].

However, the ultimate speed of the imaging techniques considered so far is limited by both technical and physiological problems. Modern MR scanners work at technical limit with field and gradient strengths just medically justifiable as rapidly switched fields can lead to the stimulation of the peripheral nerve system. By reason of patient protection, gradient strength and slew rates must not exceed certain limit values.

In order to overcome these limitations, new reconstruction techniques have been developed including the so called *Parallel Acquisition Techniques (PATs)* that allow for *Parallel MRI (pMRI)*. PATs are not a new imaging sequence but an elaborate image reconstruction technique in order to accelerate any

existing imaging sequence without changing the scanner's gradient system performance. Moreover, it is applicable to (almost) any MRI sequence and maintains its contrast behavior [27]. The general idea of PATs is to simultaneously use several receivers aligned in an *array* (*phased array coils*). The spatial sensitivity information of the coil array can evidentially be used to undersample k-space, which is equivalent to omitting some time-consuming spatial encoding steps. Therefore, the whole image acquisition procedure is commensurately reduced by the so called *acceleration factor also known as* PAT factor at the expense of more complex and time-consuming reconstruction algorithms. For the sake of completeness, it should be mentioned that this is only possible for coils that meet certain requirements concerning their sensitivity [3].
As a result of the improved time resolution, the following advantages of PATs can be distinguished:

- shortened breath-hold periods (respiratory movement)
- decreased *specific absorption rate (SAR)*
- allows for new clinical applications:
  interventional MRI, head, thoracic, and (real-time) cardiac imaging
- reduced artefacts by reason of time-dependent effects
- reduced amount of contrast agent in angiographic applications
- alternatively: improved spatial resolution (at same acquisition time)

The undersampled k-space yet leads to fold-over distortions (aliasing) outside the coils' *Field of View (FOV)*. It is the reconstruction algorithm's task to "unfold" these images with limited FOV and to combine them to an overall FOV image. This reconstruction is done by interpolation relying on precomputed spatial and sensitivity information of the coils. The precalculation can either be done by a separate scan before the actual examination or by *autocalibration* which results in better image quality besides many other advantages. Unfortunately, PATs reduce the *Signal-to-Noise Ratio (SNR)* by the expected factor $\sqrt{R}$, where R represents the acceleration factor [3].
Current pMRI techniques can be divided into three groups according to their domain of interpolation:

1. Interpolation of image data in image domain (*SENSE*, *PILS*)
2. Interpolation of raw-data in k-space (*SMASH*, *GRAPPA*)
3. Hybrid techniques (*SPACE RIP*) [42]

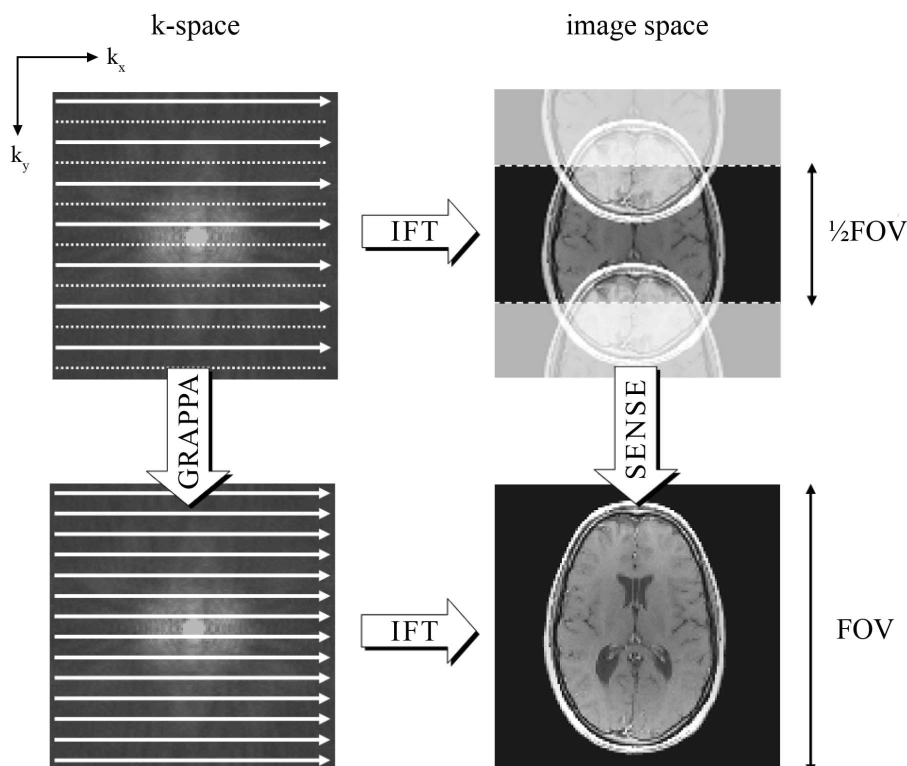


Figure 2.12: Basic pMRI Concepts
The number of required phase encoding steps is reduced by omitting some k-space lines (dotted lines in upper left image) causing aliasing artifacts in the inversely Fourier-Transformed image (top right). The GRAPPA algorithm performs image reconstruction in k-space (bottom left), whereas SENSE operates in the image domain to solve for the unfolded full FOV image (bottom right) [9].

2.5.2 Reconstruction in Image Domain

For now, the coils are assumed to be regularly aligned in a linear array along PE direction and to possess a limited FOV as well as spatially localized sensitivities. As mentioned before, the accelerated pMRI acquisition inevitably leads to aliasing in the reconstructed images. The images of each coil show repeated subimages that can be separated, however, as long as the limited FOV is larger than the region of sensitivity. The position of each coil in the array makes it possible to easily recover the lost spatial information of the proper subimage. Finally, the combination of the relocated subimages results in the desired full FOV image. This reconstruction technique is also known as *Partially Parallel Imaging with Localized Sensitivities (PILS)* and achieves optimal SNR [3].

Restrictions of PILS, particularly concerning the coil configuration with localized sensitivities, are too tight for clinical applications. A more generalized method was proposed by Pruessmann et al. in 1999: *Sensitivity Encoding (SENSE)* [71] allows for more freedom in coil configuration and sensitivities that are no longer necessarily strictly localized. Therefore, the coil images with reduced FOV now contain sensitivity weighted information of R equidistantly distributed images assuming Cartesian-type sampled k-space [9].

Given the full FOV image ρ and a set of N_c coils ($N_c \geq R$) with known individual sensitivities C_k, the signal intensity at a specific pixel location (x,y) in the limited FOV image I_k of coil k can be expressed as follows [3]:

$$I_k(y,x) = \sum_{l=1}^{R} C_k(y_l,x)\rho(y_l,x) \quad .$$

For a fixed pixel location (x,y) this results in N_c equations that can be written in matrix notation as shown in Figure 2.13.

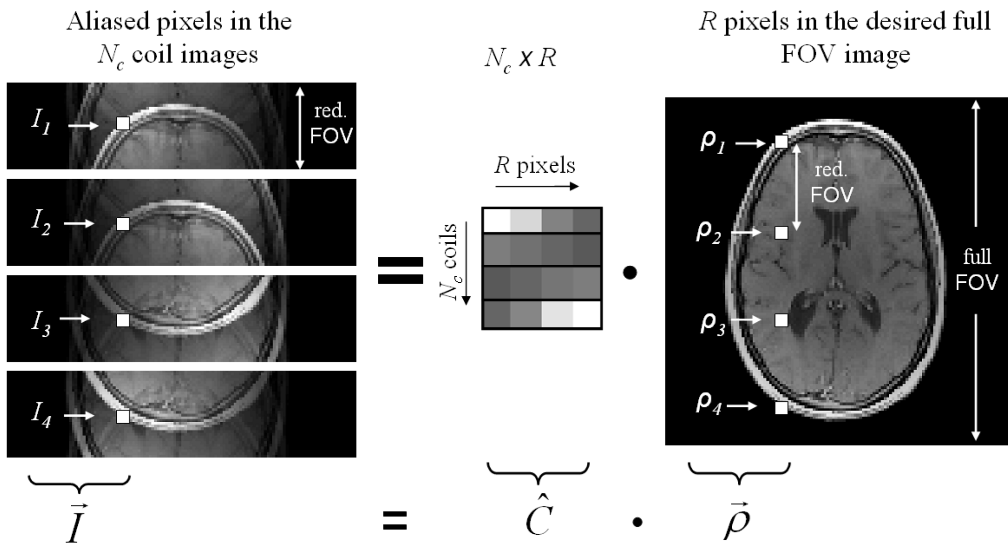


Figure 2.13: SENSE Reconstruction Scheme in Matrix Notation

If the sensitivity matrix $\hat{C}$ is known in advance, this system of linear equations can be solved by applying the pseudo inverse of $\hat{C}$:

$$\vec{\rho} = (\hat{C}^H \hat{C})^{-1} \hat{C}^H \cdot \vec{I} \,, \tag{2.2}$$

where $\hat{C}^H$ denotes the *adjoint matrix*[2] of $\hat{C}$. The computation is repeated for every pixel in the N_c aliased coil images, which finally results in the desired full FOV image. The maximal reduction factor R is N_c as the matrix inversion in Equation 2.2 would fail for larger values of R.
Regrettably, SENSE involves a worse SNR compared to PILS as it is additionally reduced by the so called *geometry-factor* $g \geq 1$ depending on the particular coil configuration [71]:

$$\mathrm{SNR}_{\mathrm{SENSE}} = \frac{\mathrm{SNR}_{\mathrm{full}}}{g \cdot \sqrt{R}} \quad .$$

Moreover, there are several enhancements such as *mSENSE* [80] or a generalization of SENSE that allows for data to be sampled along arbitrary k-space trajectories [70].

2.5.3 Reconstruction in k-space

Instead of performing the interpolation of missing information in the image domain, it is possible to recover omitted lines in k-space before the image reconstruction. One method that is based on this consideration is the *Simultaneous Acquisition of Spatial Harmonics (SMASH)* introduced by Sodickson and Manning in 1997 [77]. According to the Fourier theory, composite sensitivity profiles C_m^{comp} with sinusoidal spatial variations of order m are required to emulate the omitted phase-encoding steps. These profiles are generated by a linear combination of previously estimated coil sensitivities $C_k(x,y)$ [3]:

$$C_m^{\mathrm{comp}}(y,x) = \sum_{k=1}^{N_c} w_k^{(m)} \cdot C_k(y,x) \cong e^{im\Delta k_y y} \,, \qquad \text{where} \quad \Delta k_y = \frac{2\pi}{\mathrm{FOV}} \quad . \tag{2.3}$$

The interpolation weights $w_k^{(m)}$ can be calculated by a least square fit. This set of weights is finally used to derive composite shifted k-space lines $S^{\mathrm{comp}}(k_y + m\Delta k_y)$ from simultaneously measured coil signals $S_k(k_y)$:

$$S^{\mathrm{comp}}(k_y + m\Delta k_y) = \sum_{k=1}^{N_c} w_k^{(m)} \cdot S_k(k_y) \quad .$$

[2] The operator $\bullet^H$ is also known as *conjugate* or *Hermitian transpose*.

Furthermore, the resulting image quality highly depends on the accuracy of the generated spatial harmonics in Equation 2.3.

The coil sensitivities can also be derived by autocalibration as it is done in *AUTO-SMASH* [39]. Additionally acquired $R-1$ autocalibration lines S_k^{ACS} in the center of the k-space are used to calculate the reconstruction coefficients:

$$S^{\mathrm{comp}}(k_y + m\Delta k_y) = \sum_{k=1}^{N_c} S_k^{\mathrm{ACS}}(k_y + m\Delta k_y) \cdot S_k(k_y) = \sum_{k=1}^{N_c} w_k^{(m)} \cdot S_k(k_y) \quad . \tag{2.4}$$

It is even possible to acquire more ACS lines than actually needed in order to reduce the influence of noise and coil profile imperfections. This is done in *Variable-Density AUTO-SMASH (VD-AUTO-SMASH)* [26] and results in a more robust image reconstruction.

Another SMASH-type reconstruction technique is *GRAPPA* that is discussed in the next section in more detail as it is an essential part of this thesis.

2.6 The GRAPPA Algorithm in Detail

The concept of *Generalized Autocalibrating Partially Parallel Acquisitions (GRAPPA)* was introduced by Griswold in 2002 [22] as a more general view of (VD-)AUTO-SMASH and yields a better overall image quality due to improved artifact suppression. The most significant difference is the way missing k-space lines are reconstructed. Figure 2.14 outlines the entire GRAPPA reconstruction that is described in the following.

2.6.1 Reconstruction of k-space

In contrast to (VD-)AUTO-SMASH, the GRAPPA reconstruction recovers the missing k-space lines for all channels, which results in an uncombined image per coil and not only a single composite signal as shown in Equation 2.4. The reconstruction of a single missing k-space element within a coil is based on several blocks of k-space data from *all* channels. As shown in Figure 2.15, a *block* is defined as a single acquired element followed by $(R-1)$ omitted entries in the same image column, i.e. with

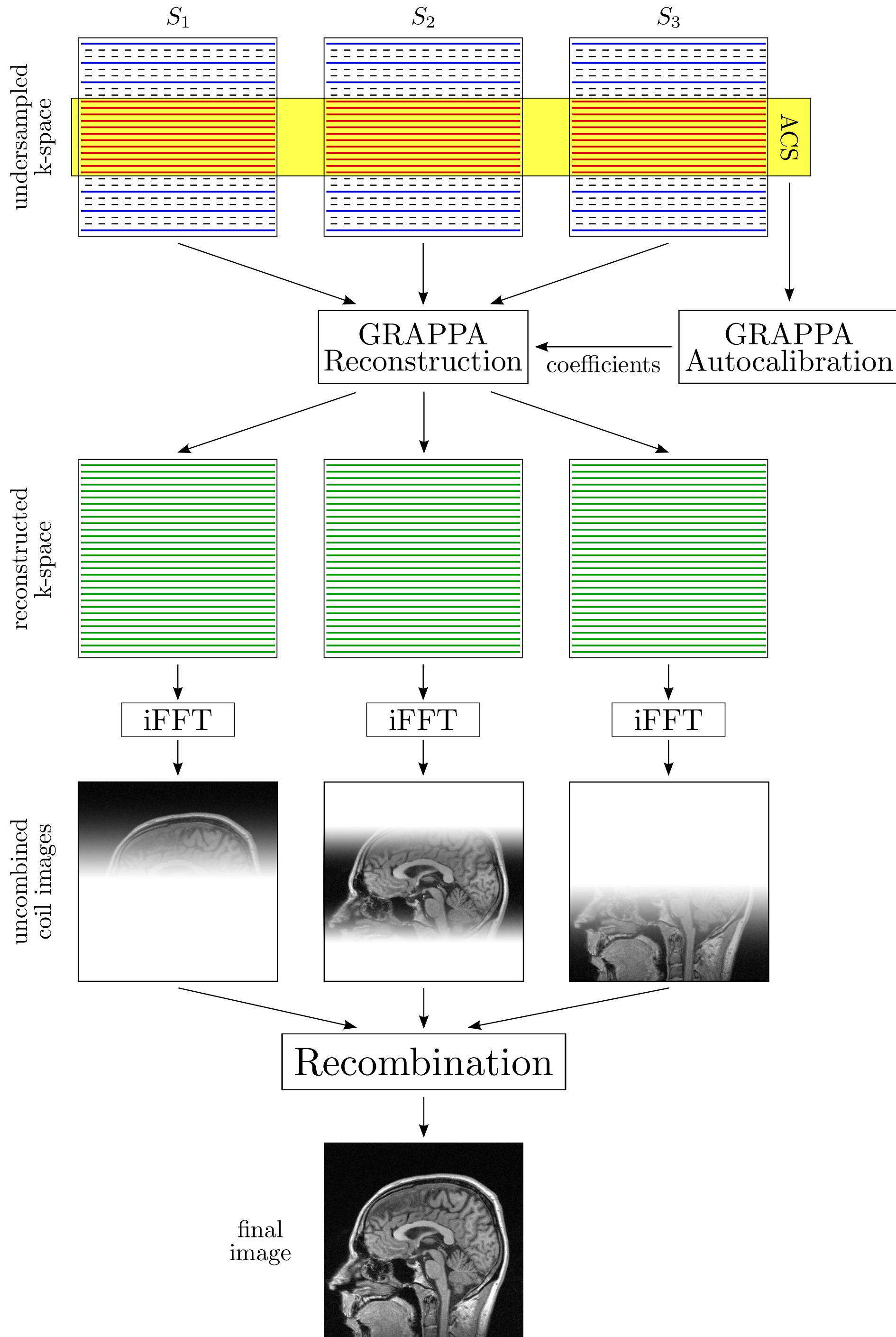


Figure 2.14: GRAPPA at a Glance
Based on the *Auto-Calibration Signal (ACS)*, the autocalibration stage computes fitting coefficients that are used to reconstruct omitted k-space lines (dashed) for each coil ($N_c = 3$ in this case). The inverse Fourier Transform of the reconstructed k-space generates a set of uncombined images that are recombined to the final slice image.

same RO-coordinate. Reconstructing an entry in line $k_y + m\Delta k_y$ and column x of target coil j can be represented as [22]:

$$S_j(k_y + m\Delta k_y, x) = \sum_{k=1}^{N_c} \sum_{b=0}^{N_b-1} w_{j,b,k,x}^{(m)} S_k(k_y + bR\Delta k_y, x) , \tag{2.5}$$

where R represents the acceleration factor and N_b the predefined number of consecutive blocks to use for reconstruction. The total number of coils is denoted by N_c, the fitting coefficients by w, and the simultaneously measured signals by S. Usually, the *filter mask* "symmetrically" surrounds the omitted element to recover. In other words, the choice of $m \in \mathbb{N}$ is restricted to

$$\underbrace{\left(\lfloor \frac{N_b}{2} \rfloor - 1\right) R}_{\Delta_{\text{ACS}}} < m < \lfloor \frac{N_b}{2} \rfloor R \ .$$

Using more than one block for the reconstruction ($N_b > 1$) allows to incorporate more information into each reconstructed line, which substantially improves the fitting. Theoretically, it is even possible to make use of all acquired blocks, which would result in an exact, SENSE-like reconstruction. Practically, however, only a few blocks (usually four to eight) close to the corresponding missing lines are used in order to reduce the computational effort but yet leading to reasonable results [77].

Besides the symmetric reconstruction scheme, it is possible to use a *sliding block technique* [22]. In this case, each missing k-space element is reconstructed based on several different configurations of surrounding reference lines. Each configuration, thus, leads to a preliminary result. The weighted combination of these estimations ultimately generates the final value of the omitted entry.

2.6.2 Autocalibration

The *autocalibration stage* uses a least-squares technique to calculate the fitting coefficients required for the GRAPPA reconstruction. The center of k-space is sampled at *Nyquist rate*, whereas the sampling rate at *outer k-space* is reduced by the acceleration factor R. The additionally acquired ACS lines in the center region are used to estimate the coil weights containing the coil sensitivity information.

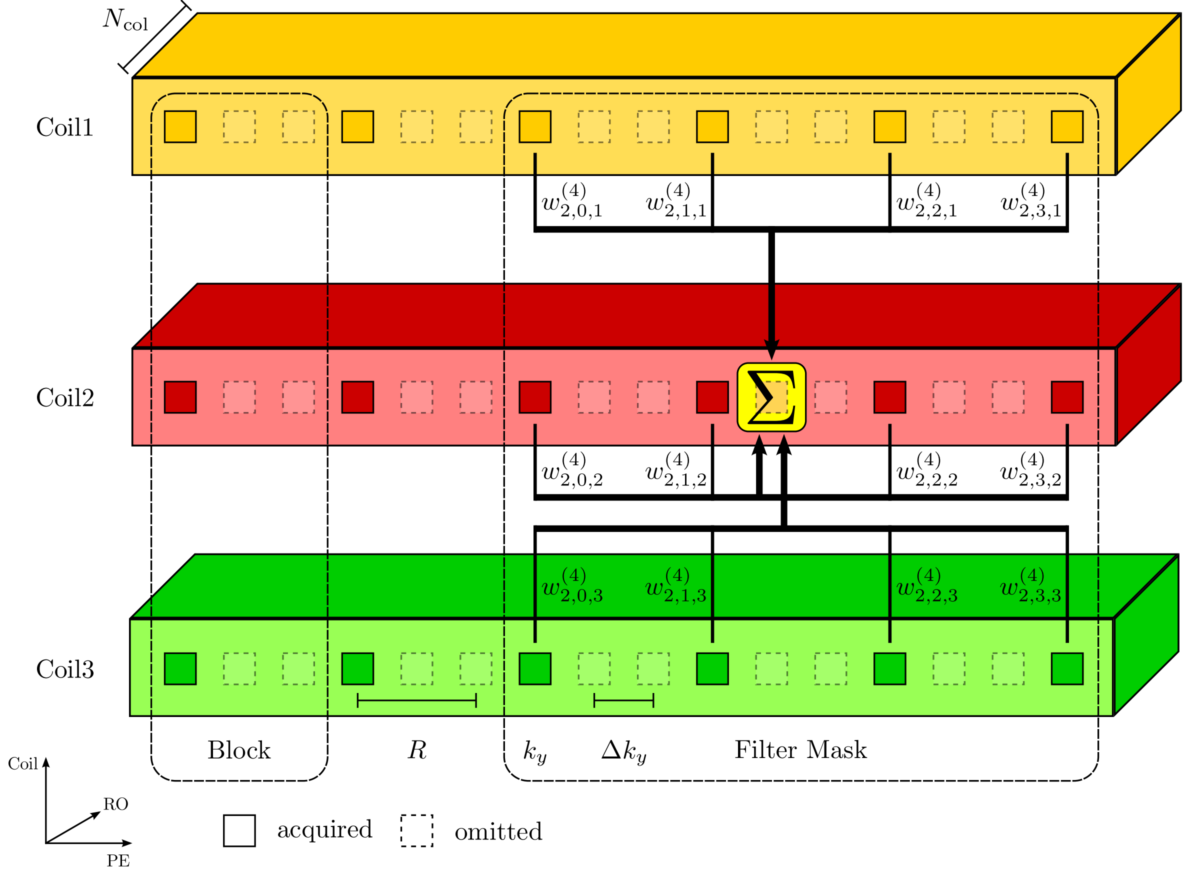


Figure 2.15: GRAPPA Reconstruction Scheme
for $S_2(k_y + 4\Delta k_y)$ (highlighted omitted element in Coil2) with $N_c = 3$, $R = 3$, and $N_b = 4$. Neighboring reference lines are weighted with proper coefficients ($w_{2,b,k}^{(4)}$) and finally added up according to Equation 2.5.

The estimation of the reconstruction coefficients follows the same scheme that is used for GRAPPA reconstruction according to Equation 2.5. The only difference is that there are no omitted ACS entries but the weights w are unknown:

$$S_j^{\text{ACS}}(k_y + m\Delta k_y, x) = \sum_{k=1}^{N_c} \sum_{b=0}^{N_b-1} w_{j,b,k,x}^{(m)} S_k^{\text{ACS}}(k_y + bR\Delta k_y, x) \ . \tag{2.6}$$

The autocalibration algorithm slides the filter mask shown in Figure 2.15 over all ACS lines along PE-direction. Given the overall number N_r^{ACS} of acquired reference lines, there are

$$N_b^{\text{ACS}} = N_r^{\text{ACS}} - (N_b - 1)R$$

different filter mask positions within the ACS data. The fitting process described in Equation 2.6 is repeated for every coil and for all N_b^{ACS} filter masks to incorporate as much information as possible into the computation of the reconstruction weights.

Furthermore, it is possible to partition the k-space along the RO-direction into several *segments* of length Δs. As the sensitivities of each coil are assumed to be approximately constant within each segment, the fitting coefficients have to be computed only once per segment (see Section 4.1).
All in all, this results in a large (over-determined) system of linear equations for each segment of the following form:

$$\underbrace{\left[B_{(k_{y_0}+i\Delta k_y)}\right]_{0\leq i<N_b^{\text{ACS}}}}_{B} = W \cdot \underbrace{\left[A_{(k_{y_0}+i\Delta k_y)}\right]_{0\leq i<N_b^{\text{ACS}}}}_{A}, \tag{2.7}$$

where k_{y_0} is the position of the first ACS block,

$$A_{k_y} := \begin{pmatrix} \left[S_1(k_y+bR\Delta k_y)\right]_{0\leq b<N_b} \\ \vdots \\ \left[S_{N_c}(k_y+bR\Delta k_y)\right]_{0\leq b<N_b} \end{pmatrix} \in \mathbb{C}^{[N_b\cdot N_c]\times 1},$$

$$B_{k_y} := \begin{pmatrix} \left[S_1(k_y+(\Delta_{\text{ACS}}+i)\Delta k_y)\right]_{1\leq i<R} \\ \vdots \\ \left[S_{N_c}(k_y+(\Delta_{\text{ACS}}+i)\Delta k_y)\right]_{1\leq i<R} \end{pmatrix} \in \mathbb{C}^{[(R-1)N_c]\times 1},$$

$$W := \begin{pmatrix} \left[w_{1,b,1}^{(m)}\right]_{1\leq m<R}^{0\leq b<N_b} & \cdots & \left[w_{1,b,N_c}^{(m)}\right]_{1\leq m<R}^{0\leq b<N_b} \\ \vdots & \cdots & \vdots \\ \left[w_{N_c,b,1}^{(m)}\right]_{1\leq m<R}^{0\leq b<N_b} & \cdots & \left[w_{N_c,b,N_c}^{(m)}\right]_{1\leq m<R}^{0\leq b<N_b} \end{pmatrix} \in \mathbb{C}^{[(R-1)N_c]\times[N_b\cdot N_c]},$$

$$\left[w_{j,b,k}^{(m)}\right]_{1\leq m<R}^{0\leq b<N_b} := \begin{pmatrix} w_{j,0,k}^{(1)} & \cdots & w_{j,(N_b-1),k}^{(1)} \\ \vdots & \cdots & \vdots \\ w_{j,0,k}^{(R-1)} & \cdots & w_{j,(N_b-1),k}^{(R-1)} \end{pmatrix}.$$

The step size in RO-direction is denoted by Δk_y.

The system can be solved for W by applying the *Moore-Penrose-Inverse* also known as *pseudo-inverse* of A denoted by A^+ [17]:

$$A^+ := \lim_{\lambda \to 0} \left(A^H (AA^H - \lambda E)^{-1} \right) \tag{2.8}$$
$$W \cong BA^+ \cong \left(BA^H \right) \left(AA^H - \lambda E \right)^{-1} ,$$

where E denotes the identity matrix. Please note that λ is a constant in the practical application and has to be chosen very carefully to obtain reasonable results.

Needless to say, ACS lines that have been used for the autocalibration stage can directly be incorporated into the reconstructed k-space image to further improve image quality.
After the reconstruction of all missing k-space lines for each coil, the inverse Fourier Transform generates an uncombined image for each coil. There are different possibilities how to eventually join the full set of uncombined images to the final slice image such as a standard sum of squares approach.

2.6.3 Further Development

The GRAPPA algorithm is all-important in practice and has been refined again and again. We give a short overview of recent enhancements.
Huo and Wilson for instance came up with *Robust GRAPPA* in 2006 [35]. They adjusted the calculation of the reconstruction coefficients by reducing the influence of outliers within the calibration region. In this way, the fitting accuracy and the overall quality of reconstructed images are improved at the expense of doubled computational effort compared to conventional GRAPPA.
A further GRAPPA-based approach was proposed by Blaimer in order to accelerate 3D pMRI [4]. His *2D-GRAPPA Operator* reduces the number of necessary phase-encoding steps in two spatial dimensions requiring two separate reconstruction passes, one for each dimension.
Another field of research is the combination of different reconstruction techniques to improve image quality. Hoge and Brooks developed a hybrid technique that leverages the advantages of both GRAPPA and SENSE [31]: GRAPPA performs better in estimating low-frequency components of k-space, whereas SENSE is better suited for high-frequency components.

Blaimer et al., finally, combined the 2D-GRAPPA operator with SENSE for accelerated volumetric MRI [5].

A great deal of research has also been done in *dynamic pMRI*. Several *image frames* are generated by a combination of GRAPPA and a *time-interleaved acquisition scheme*. This process is called *Temporal GRAPPA (TGRAPPA)* [10]. For coil calibration, directly adjacent time-frames are merged to generate a set of completely encoded full-resolution reference data. The need for acquiring additional reference data is thus eliminated. Moreover, they showed that coil coefficients and sensitivity estimates can be updated dynamically frame-by-frame, which allows for tracking changes during the acquisition process.
Huang et al. subsequently introduced a technique similar in design: *k-t-GRAPPA* [34]. As the name suggests, the main idea of GRAPPA is applied in both k-space and t-space.

2.7 Imaging Hardware

An MR system consists of several parts:

The magnet is the part and parcel of an MR system. It produces a static, homogeneous, and very strong magnetic field of $0.5 - 3.0\,\mathrm{T}$ for clinical applications and up to $8\,\mathrm{T}$ respectively for research purposes. A high quality grade and a high field strength significantly improve the image quality of the resulting MR images by a better SNR [46, 29]. This is the reason why costly manufactured magnets are used in modern MR systems to make the grade. For high-field systems ($B > 0.5\,\mathrm{T}$), *superconducting magnets* are commonly used. Superconductivity persists only at very low temperatures close to absolute zero ($0\,\mathrm{K} = -273\,^{\circ}\mathrm{C}$). Therefore, the magnet is cooled with a *cryogenic cooling fluid*, usually liquid helium. Once, the super-conducting magnet is energized to the desired field strength, it persists without any further current supply as long as the cooling system works properly.
Besides superconducting magnets, it is also possible to use other types of magnets, e.g. permanent magnets and resistive electromagnets. Magnets of this kind are usually only capable of lower magnetic field intensities ($B < 1.0\,\mathrm{T}$) compared to superconducting magnets.

Moreover, it is possible to distinguish between open and closed bore magnets. The former generates a vertically directed main magnetic field, the latter a horizontally directed one.

Patients equipped with ferromagnetic implants such as pacemakers have to be excluded from MR examinations as the strong magnetic field would cause serious injury.

Radiofrequency coils

Transmitter coils generate the sequence of RF pulses to stimulate the nuclear spins of the body tissue (see Chapter 2.2.3). They surround the whole, or a part of the patient's body.

The resulting MR signal, which provides the diagnostic information, is detected by *receiver coils*. As the signal is very weak and sensitive to electrical interference, special shielding (*Faraday Cage*) is required for the room in which the MR scanner is installed. Furthermore, there are special receiver coils of different size and shape determined by its respective area of application, e.g. coils for head, spine, breast, etc.

Gradient coils are required for slice selection as well as frequency and phase encoding (see Chapter 2.3). There are three sets of gradient coils to achieve arbitrary gradients in 3D, one set for each direction. Usually, they are installed in the magnet's bore. These coils generate a continual, rapidly hammering noise during a scan. The loudness of this sound may even exceed safety guidelines and thus requires ear protection.

Computer Systems

The operator prescribes the desired pulse sequence with its corresponding parameters for an examination on a host computer. These parameters are then passed to multiple subsystems with their own microprocessors that are responsible for proper device control. Another computer system, then, processes the acquired data and generates a set of reconstructed MR images that can finally be archived, printed, or simply visualized on the host computer.

Some more technical details of MR hardware are given in [46] and [49].

Chapter 3

General-Purpose Computing on GPUs

3.1 Introduction

Recent desktop PCs are all equipped with more and more powerful *Graphics Processing Units (GPUs)* originally designed for compute-intensive and highly parallel computations primarily in the range of computer graphics. Over the past years, the ever-growing distribution of GPUs has created a new area of research trying to use computer graphics hardware for non-graphics applications. Meanwhile, there has even emerged a whole community around *General-Purpose Computing on GPUs (GPGPU)* [24].

3.1.1 Why GPUs for General-Purpose Computing?

GPUs are suitable for a general-purpose use for various reasons:

First and foremost, GPUs provide a highly specialized compelling platform for computationally intensive tasks. In recent years, there has been a substantial increase in *performance* owing to - among other factors - the game industry. The growth rate of GPU peak performance significantly outpaces the well-known Moore's law for CPUs as visualized in Diagram 3.1.

The reason for this *new Moore's law* is the specialized design of GPUs. They are optimized for high throughput and arithmetic intensity using a tremendous number of floating-point units, whereas CPUs are optimized for low latency. Therefore, additional transistors on GPUs can be devoted to computation and data processing rather than data caching and sophisticated flow control, which results, by the way, even in a better energy efficiency. In other words, the computational power of GPUs dwarfs that

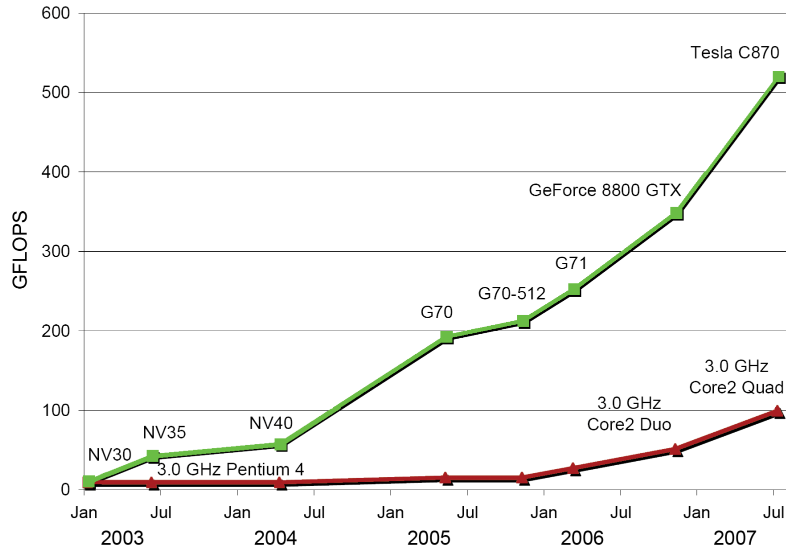


Figure 3.1: Trend of Peak Performance Since 2003

	GPU (NVIDIA)		CPU	
	GeForce 8800 GTX	Tesla C870	Intel Core2 Duo E6700	Quad-Core Intel Xeon X5482
Computing Elements	16 multiprocessors (8 ALUs each at 1.35 GHz)		2 cores	4 cores
Core Clock	575 MHz		2.66 GHz	2.88 GHz
Theoretical Total Peak Performance	345 GFLOP/s		43 GFLOP/s	92 GFLOP/s
Dedicated Memory	768 MB (GDDR3)	1.5 GB (GDDR3)	depending on additional hardware	
Bandwidth	86.4 GB/s	76.8 GB/s		
Price	$ 500	∼ $ 1000	$ 250[a]	$ 800[a]

[a]without memory

Table 3.1: Comparison of Recent GPUs and CPUs

of powerful recent *Central Processing Units (CPUs)* of high-end server CPUs as shown in Table 3.1. Another advantage is the cost-effectiveness since GPUs feature a much better cost-performance ratio compared to CPUs. Ever since the programmability of this affordable and high-performance hardware is provided[1], the deployment of GPUs for general-purpose suggests itself, of course.

[1]The programmability of GPUs has been improved over and over again with respect to a more and more user-friendly design especially in recent years (see Chapter 3.2 and 3.3).

3.1.2 Limitations and Requirements

GPGPU is in no way a panacea, though. Due to the massively parallel and specialized architecture, not all problems are well suited for GPUs such as those that are inherently sequential or cause unpredictable memory access as scatter memory operations[2] are rather slow especially for uncoalesced access patterns [67]. GPU-based algorithms have to use inherent pipelining, parallelism, and *Single Instruction, Multiple Data (SIMD)* capabilities, along with the provided vector-processing functionalities [45] to run efficiently on a GPU and harness its computational power. Moreover, we have to be aware of (slightly) limited memory resources on a GPU as well as the low bandwidth between *host* (CPU) and *device* (GPU). Applications with high *arithmetic intensity*[3] streaming through large quantities of data are ideal as memory access is relatively slow compared to arithmetic operations. As GPUs are only capable of 32-bit single-precision arithmetic at present, they are not applicable to very large-scale computational problems [67]. However, the leading manufacturers *AMD* and *NVIDIA* announced the release of GPUs supporting double-precision (64-bit) *Floating-Point (FP) numbers* for 2008.

3.2 Programmability of GPUs

As already mentioned, graphics cards have originally been designed for (interactive) computer graphics. The rapid rasterization of geometric primitives up to 60 times per second for real-time graphics such as videogames was and still is the main task which is put into practice by pipelining.

3.2.1 Graphics Pipeline

First of all, each scene to be rendered has to be described by geometric primitives possibly along with additional information concerning light sources, material characteristics and the viewer's position. The decomposition results in a stream of primitive *vertices* and *fragments* that are processed by the *Graphics Pipeline* that is sketched in Figure 3.2. This pipeline consists of different stages that are mostly implemented in hardware to reach maximum throughput. In the very beginning, the functionality of the graphics pipeline was fixed and no programmability was exposed to programmers. Nevertheless,

[2] Write access to arbitrary memory location i: $\mathrm{out}[i] = \mathrm{value}$

[3] number of arithmetic operations per data word

this feature came along over the years and provided an improved flexibility of the pipeline due to programmable *vertex*, *geometry*, and *fragment shaders*. Since then it has been possible to use GPUs not only for image-synthesis, but to consider them as computational coprocessors. This was the beginning of the GPGPU era.

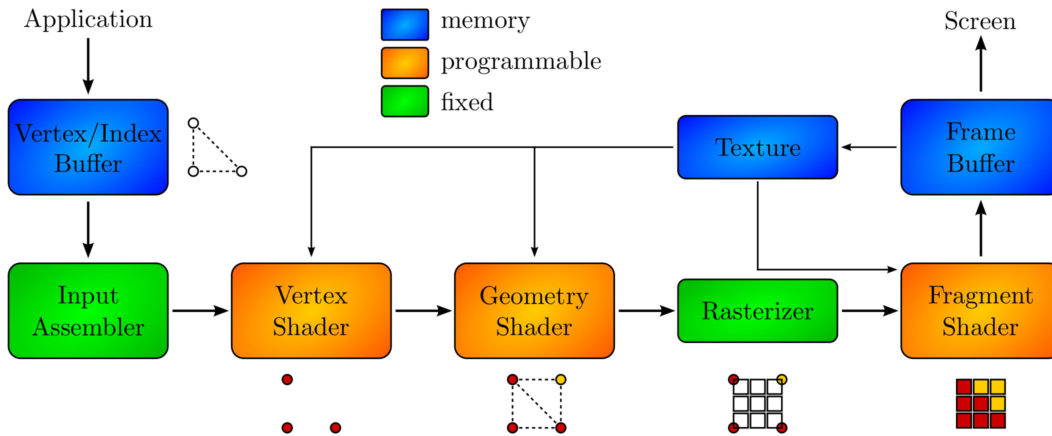


Figure 3.2: Graphics Pipeline
The pipeline processes a stream of geometric primitives in different stages. Geometry shaders have been introduced in DirectX 10 as an additional stage that allows to generate new geometric primitives such as points, lines, and triangles without CPU intervention.

3.2.2 High-Level Languages

Meanwhile, various different *Application Programming Interfaces (APIs)* and corresponding *shading languages* have been invented by different industry foundations and the manufacturers itself in order to facilitate and standardize the programming of graphics hardware. There are essentially three shading languages that have become widely accepted in the beginning as listed in Table 3.2. These languages

Language		Used Standard	Inventor	
GLSL	OpenGL Shading Language	OpenGL	ARB[a]	[65]
HLSL	High Level Shading Language	DirectX	Microsoft Corp.	[47]
Cg	C for graphics	OpenGL/DirectX	NVIDIA Corp.	[54]

[a]OpenGL Architecture Review Board

Table 3.2: GPU Shading Language Overview

all provide almost the same functionality that can be used for GPGPU. There is a recommendable "Basic Math Tutorial" by Dominik Göddeke [16] explaining how to use the special features and the unusual programming model of graphics cards for a simple linear algebra operator. Computing on graphics hardware is like drawing. Therefore an algorithm initially has to be transformed in terms of graphics. From a programmer's point of view, GPGPU is not only a matter of simply learning a new programming language but understanding a completely different concept of programming. The strength of these shading languages lies definitely in real-time graphics but other tasks are rather cumbersome. The reasons therefore are limited shader capabilities:

- restricted addressing modes (lack of *scatter operations*)
- limited instruction set (integer and bit operations not supported until recently)
- limited communication and information exchange between pixels

In recent years, though, new graphics hardware and many high-level and meta-programming frameworks have been developed providing more flexibility and an even higher level of abstraction, which allows to rather focus on the algorithm than on implementation tasks. Houston overviewed "High Level Languages for GPUs" [33] which are partly shown in Table 3.3.

Language	Inventor	
Accelerator	Microsoft Research	[48]
Brook	Stanford University	[11]
CTM[a]	Advanced Micro Devices, Inc.	[1]
CUDA[b]	NVIDIA Corp.	[63]
RapidMind	RapidMind, Inc.	[73]

[a] Close To The Metal

[b] Compute Unified Device Architecture

Table 3.3: Overview of GPGPU High-Level Languages

3.3 Compute Unified Device Architecture

As this thesis focuses on NVIDIA's *Compute Unified Device Architecture (CUDA)*, the main aspects of this recent software environment are explained in this section. Further particulars can be found in the *CUDA Programming Guide* [61] and in the *CudaZone* [63].

3.3.1 Overview

With their latest G80 series chips launched in November 2006, NVIDA embarked on a new strategy. They moved away from the long-established graphics pipeline to a more flexible general-purpose computational engine with *Unified Shaders*. On previous graphics hardware, there are separate custom processing units for vertex and fragment shaders. The unified shader architecture, however, uses several data-parallel FP processors that can run each of the shaders. As opposed to the usual vertex/fragment processor architecture, it prevents from imbalanced workload and therefore results in an improved overall utilization. This imbalance may be caused for instance by unfavorable scenes. Simply lighted scenes with complex geometry lead to vertex processors used to full capacity but under-worked fragment processors. Correspondingly, we get the contrary result for scenes with rather simple geometry but complex illumination.

CUDA provides a completely new environment and programming model for GPGPU and goes without conventional graphics APIs such as OpenGL or DirectX. Instead, the programmer has the possibility to use either the *CUDA Driver API* or the *CUDA Runtime API*. The former offers more flexibility, whereas the latter makes life a lot easier without loss of efficiency for the most part. There is even a third API layer at the programmer's command comprising two self-contained higher-level libraries *CUBLAS* [56] and *CUFFT* [57] implementing a subset of core *Basic Linear Algebra Subprograms (BLAS)* and the *Fast Fourier Transform (FFT)*.

3.3.2 Programming Model

The applications themselves are implemented using a programming language that is defined by the CUDA API. As it is based on *ANSI C*, with only minimal extensions for concurrency where necessary, it is very easy to learn from a programmer's point of view. Basically, there are two different types of source code within a program. Some parts are executed on the host, others are simultaneously carried out on the device. The source code includes both host side code and function calls of special CUDA API routines as well as potential user-defined CUDA device functions also referred to as *kernel (program)*. The *NVIDIA C compiler (nvcc)* [55] available in the *CUDA Toolkit* [63] preprocesses and separates the hybrid code into CPU code and intermediate NVIDIA Assembly, also known as *Parallel Thread Execution Code (PTX)* [60]. The host side code is then compiled by a standard C compiler

whereas the PTX code is translated to target code with respect to the underlying hardware and target specific optimizations.

It is the host program's task to initiate and coordinate the function calls to the device. The general course of action is as follows:

1. Prepare and initialize input data
2. Upload input data (and instructions[4]) from host to device
3. Perform calculations on device by executing kernel programs
4. Download results from device to host.

In contrast to former approaches, CUDA and the underlying hardware now also support full integer and bitwise instructions, unlimited branching and looping, as well as arbitrary scatter and gather operations. In terms of efficiency and performance, however, we should handle these tools for flow control and memory access with care and take into account some basic rules listed in Section 3.3.7.

3.3.3 Execution Model

GPUs are equipped with a large number of computational units (see Chapter 3.3.5). Each kernel is executed concurrently by thousands of light-weight threads on different data. Threads are grouped into *thread blocks* of the same size per kernel (up to 512). Each block is processed by a single G80 *Multiprocessor (MP)* that is capable of simultaneously processing a maximum number of 8 blocks. The smallest subgroup of threads within a block that is executed physically in parallel in a SIMD fashion is denoted as *warp*. The *warp size*, i.e. the number of threads in a warp, is 32 on G80 at present. As the warps of a block are themselves executed logically parallel in an undefined execution order (time-sliced scheduling), CUDA provides special routines to synchronize thread execution per block. Threads of one block share their data through fast *shared memory*, which allows for cooperation and inter-communication within a whole block (see also Chapter 3.3.4).

Blocks of similar type, i.e. of the same size and executing the same kernel, are aligned in a one or two dimensional *grid*. During execution, each thread has access to the corresponding *thread index* within the according block and its *block index* within the grid as described in Appendix A.4. It is particularly

[4]This is done automatically.

important to note that information exchange between threads is limited as reliable communication and synchronization between different blocks within a grid is not supported in CUDA.

All in all, this is a *Single Program, Multiple Data (SPMD)* execution model as each block in the grid executes the same kernel. Grid and block dimensions have to be specified for each kernel call as shown in Listing 3.1.

Listing 3.1: Launching a CUDA Kernel

```
dim3 block(<BLOCK_SIZE_X>, <BLOCK_SIZE_Y>, <BLOCK_SIZE_Z>);
dim3 grid(<GRID_SIZE_X>, <GRID_SIZE_Y>)

myKernel<<<grid, block>>>(<parameter1>, <parameter2>, ...);
```

3.3.4 Memory Model

The CUDA memory model is illustrated in figure 3.3. Basically, there are four different types of memory adapted for different purposes [61]:

1. Global Memory

   Main *device memory* of large capacity (up to 1.5 GB) that is accessible from all threads. As there is no implicit caching, access is very slow (between 400 and 600 clock cycles).

2. Shared Memory

   Dedicated data cache limited to 16 kB per MP (organized into 16 *banks*[5]). Access is as fast as registers with low latency. Explicitly managed by the programmer and shared between all threads within a block for inter-thread communication.

3. Constant Memory

   Read-only memory accessible from all threads but limited to 64 kB with a cache working set of 8 kB per MP.

4. Texture Memory

   Read-only region of device-memory with a cache working set of 8 kB per MP. The texture cache is optimized for 2D spatial locality and designed for streaming fetches with a constant latency. Texture memory is readable for all threads.

[5]Memory modules of 32 bit bandwidth that can be accessed simultaneously [61].

Additionally, each MP is equipped with $N_r^{\mathrm{mp}} = 8,192$ registers that are uniformly distributed among all threads of N_b^{mp} concurrently processed blocks per MP. The number N_r^t of registers available per thread is limited to [61]:

$$N_r^t = \frac{N_r^{\mathrm{mp}}}{N_b^{\mathrm{mp}} \cdot ceil(N_t^b, 32)} , \tag{3.1}$$

where N_t^b is the number of threads per block and $ceil(\bullet, k) = k \cdot \lceil \bullet / k \rceil$.

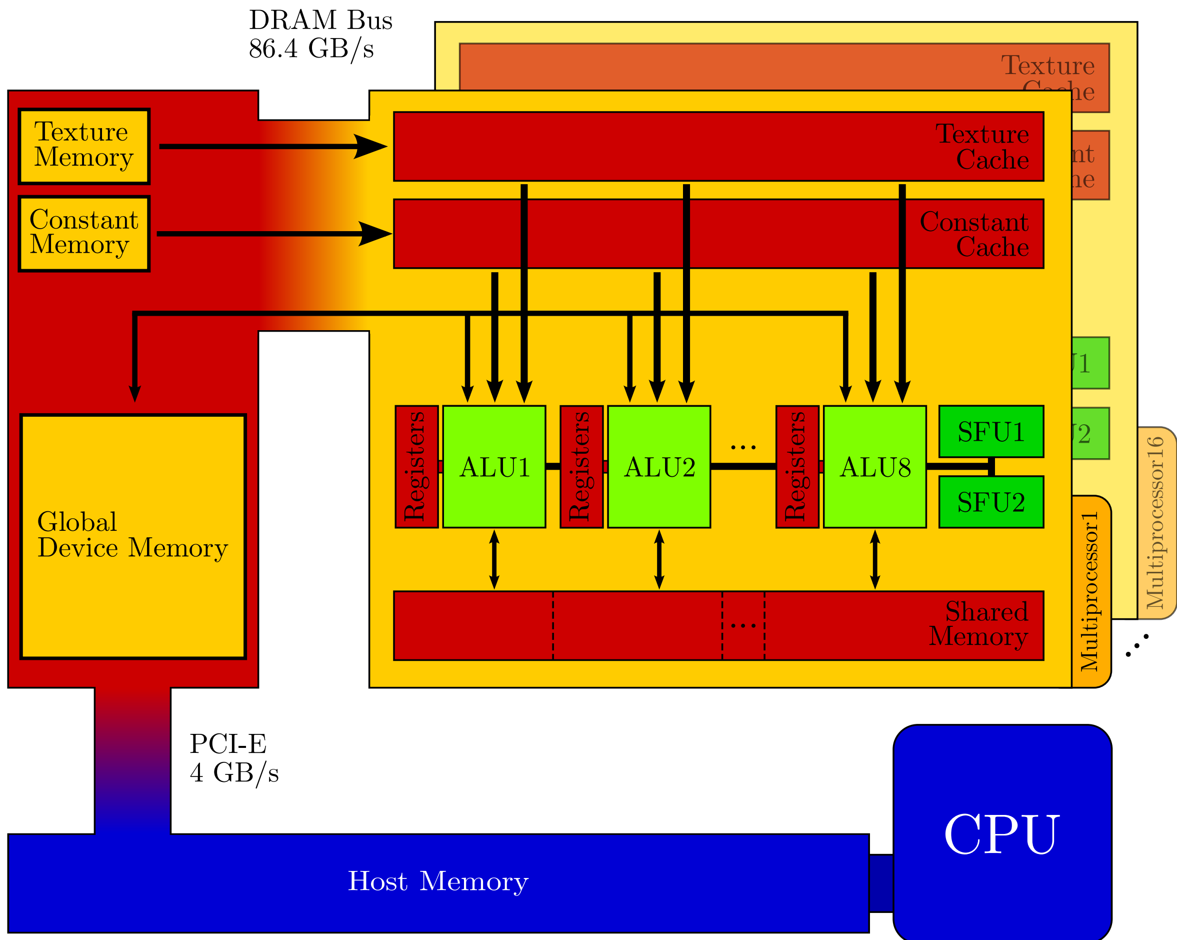


Figure 3.3: CUDA Memory Model

3.3.5 Hardware

There are several CUDA-enabled NVIDIA GPUs with different performance. The GeForce 8800 GTX is equipped with 16 MPs consisting of two *Special Function Units (SFUs)*, eight *Arithmetic Logic Units (ALUs)* clocked at 1.35 GHz, and 16 kB cache per cluster and 768 MB of device memory (see Table 3.1 on page 34).

The computation units support a full featured instruction and IEEE 754 single-precision (32-bit) FP numbers. For the sake of completeness, we should mention that arithmetic FP operations are not yet perfectly IEEE-compliant. Plans of support for double-precision (64-bit) are announced in upcoming chip sets also by AMD for 2008.

Moreover, NVIDIA offers the *High-Performance Computing (HPC)* oriented *Tesla* product line primarily used for scientific applications. These devices use up to four high-end G80 GPUs with a total of 6 GB device memory (1.5 GB each).

3.3.6 Designing Parallel Algorithms

A general "Introduction to Parallel Computing" is given by Grama [19]. The design of parallel algorithms (for any multiprocessor architecture) usually proceeds in the following steps:

1. Computations are decomposed into different *tasks* that can be executed simultaneously
2. Tasks are mapped onto *processes*[6]

There are different techniques for both steps depending on the particular problem. Moreover, the circumstances of the underlying target programming platform necessarily have to be taken into consideration.

Now, we have to translate this general scheme into the terminology of CUDA. According to Section 3.3.2, there is a static mapping of tasks onto processes, even if there is no exact equivalent to this term. Depending on the decomposition and the granularity, it may make sense to map tasks onto single threads or even onto thread blocks. As a reminder, the hierarchy of grids, blocks, warps, and threads includes different restrictions regarding resource utilization and inter-thread communication,

[6]The term "process" is used in broader sense in this case and there is no, or at least not necessarily a relation to the definition of a process in an operating system.

i.e. synchronization and shared memory access space. This has to be taken into consideration especially for the first step. Further details of the underlying GPU platform, that we have to account for, are described in the next section.

3.3.7 Performance Aspects

In order to efficiently use the massively parallel performance potential of GPUs it is of particular importance to follow some basic rules. A couple of recommendable tutorials concerning optimal CUDA performance can be found at www.gpgpu.org [24, 25, 82]. Very high speedups can be achieved only by understanding the GPU and CUDA architecture [61].

Basically, there are three different fields of optimization dealing with

1. data transfer between host and device
2. memory usage and access
3. utilization of parallelism

In the following, we highlight the most important aspects of these categories that can help to avoid common pitfalls.

Data Transfer The CUDA supporting device is connected to the host via 64-bit *PCI Express x16 Graphics Interface* with a peak bandwidth of 4 GB/s per direction [38]. As the bus bandwidth is the bottleneck, it is advisable to reduce data transfers between host and device to a bare minimum[7]. Grouping all transfer tasks to a single large data transfer minimizes the transmission time. Moreover, CUDA offers intrinsic routines to allocate *page-locked (host) memory* that remains unaffected by memory swap operations. On the one hand, pinned memory speeds up transfers, on the other hand it is yet a limited resource and has to be allocated well-considered too much of it may harm the overall system performance.

Memory Usage Particularly, the optimization of memory usage and access patterns yield highest time savings in the majority of cases.

Read and write instructions on the global memory are likely to be a performance bottleneck as a single

[7] see also general program layout in Section 3.3.2

access operation takes 400 up to 600 clock cycles. For this reason, it is recommendable to leverage shared memory with low latency and high bandwidth. A kernel usually follows the steps described below:

1. load data from global memory to shared memory
2. synchronize threads (within block)
3. process data in shared memory
4. synchronize threads (if necessary)
5. copy result(s) from shared memory to global memory

Accessing global device memory, *coalesced memory access* is crucial to maximize bandwidth. Optimized memory access patterns may result in a speedup of ten compared to non-coalesced read and write operations. In the same way, we should take care of spatial locality accessing cached texture memory.
As shared memory is organized in banks (see Section 3.3.4) it is again the programmer's task to avoid high-degree *bank conflicts*. Time savings due to bank conflict avoidance are usually rather low, though. Finally, memory latency can be hidden by scheduling different warps during memory access whenever possible.

Utilization of parallelism The massively parallel computational power of graphics hardware can only be used efficiently as long as there are enough thread blocks to keep the multiprocessors busy. To achieve this goal, the structure of the according algorithm has to be adjusted for maximized independent parallelism. Moreover, we have to partition the corresponding computations at a proper level of granularity and select "suitable" kernel execution parameters, e.g. grid and block layout. As a matter of principle, the number of blocks needed for the execution of a single kernel should be (considerably) larger than the number of available multiprocessors. For efficient latency hiding, it is worthwile to maximize arithmetic intensity but to keep resource usage (registers, shared memory) low enough to allow for several active blocks per MP (see Equation 3.1). In other words, we have to strive for high *multiprocessor occupancy*:

$$\text{Occupancy} = \frac{\text{number of warps running concurrently on an MP}}{\text{maximum number of concurrent warps on an MP}} .$$

The overall performance is, however, not necessarily augmented by increased occupancy. Nevertheless, multiprocessors with low occupancy cannot adequately hide latency for memory-bound kernels. NVIDIA provides a useful developer tool, the *NVIDIA Occupancy Calculator* [58], that supports in optimizing kernel execution configurations and occupancy. Just recently, the *NVIDIA Visual Profiler* has been released providing detailed and very helpful information about kernel behavior during execution such as execution time, degree of branching, and basic memory access pattern analyzes among many others [64].

Utilization of *control flow instructions* is yet another aspect of optimizing for parallelism. As already mentioned, the GPU multiprocessors execute warps based on the SIMD paradigm. Instructions of different traversed branches have to be serialized, which results in SIMD *divergence*. During the execution of instructions belonging to a specific branch, only threads fullfilling the appropriate branch condition can be executed, all other threads within a warp are idle. Consequently, branching instructions have to be used well-considered to minimize divergent warps.

Other possibilities There is a multitude of other techniques for performance optimization in CUDA. Even if arithmetic operations are much faster compared to memory access, they are not for free by no means. Especially, more complex operations on FP numbers such as division, square root, logarithm, exponential function, sine, and cosine take several clock cycles. This may be a potential bottleneck for computationally limited kernels. The *CUDA Runtine Math Library*, though, provides functions performing much faster but with reduced 24-bit accuracy which is still sufficient in certain cases [61]. Finally, there are many elaborate techniques which may result in tremendous speedups such as *loop unrolling*, *template programming*, or optimization of PTX code.

3.4 GPGPU in Practice

Besides medical imaging, there are various GPGPU applications in different fields of research harnessing the computational power of graphics hardware, particularly for parallel data processing and compute-intensive tasks:

- cryptography
- linear algebra
- computational geometry
- motion planning and navigation
- global illumination
- sorting
- database management and data mining
- signal processing
- physical based simulations (fluid dynamics)
- computational finance
- computational biology and bioinformatics

In many cases, GPU-based algorithms outperform CPU-based approaches by an order of magnitude. A showcase and success story for instance is the current project *Folding@Home* at the Stanford University [68]. Their goal is to study the complex process of *protein folding* and related diseases using distributed clients all over the world. In October 2006, an ATI-GPU accelerated client contributed 28,000 GFlops in one month, which is more than 18% of the total amount contributed by CPU clients since October 2000.

For further information about GPGPU applications and techniques we recommend [59, 24].

Chapter 4

GPGPU for GRAPPA Autocalibration

We have already presented theoretical details of the GRAPPA algorithm in Section 2.6. This chapter focuses on the GRAPPA autocalibration stage and presents a GPU-based approach. The initial point is the *C++* GRAPPA implementation courtesy of *Siemens Medical Solutions, Erlangen* that is applied for image reconstruction in current Siemens MR systems.

We present the design and progression of our approaches step-by-step based on pseudo code snippets. Please note Section A.3 and A.4, particularly for semantical details.

4.1 Autocalibration Algorithm in Practice

There are two different versions for the autocalibration stage. Basically, they both follow the same pattern of Listing 4.1 to compute the coefficient matrix W according to Equation 2.7.

Listing 4.1: Steps of GRAPPA Autocalibration Stage

```
1. set up matrices A and B using ACS lines
2. normalize A and B
3. U = B A^H
4. V = (A A^H - λ E)^-1
5. W = U · V
```

In the following we refer to these basic steps.

4.1.1 Basic Approach

The acquired k-space of each coil can be partitioned along RO-direction into several segments of length Δs (see Section 2.6.2). For each of these Δs columns within a segment, we get a system of linear equations according to Equation 2.7. Assuming approximately constant coil sensitivity within each segment, however, we can combine the Δs equation systems with equal W by merging the corresponding matrices A_{k_y} and B_{k_y}:

$$\hat{A}_{k_y} := \begin{pmatrix} \left[S_1\left(k_y + bR\Delta k_y,\, k_{x_0} + s\Delta k_x\right) \right]_{0 \le s < \Delta s}^{0 \le b < N_b} \\ \vdots \quad\quad \vdots \quad\quad \vdots \\ \left[S_{N_c}\left(k_y + bR\Delta k_y,\, k_{x_0} + s\Delta k_x\right) \right]_{0 \le s < \Delta s}^{0 \le b < N_b} \end{pmatrix} \in \mathbb{C}^{[N_b \cdot N_c] \times \Delta s}, \tag{4.1}$$

$$\hat{B}_{k_y} := \begin{pmatrix} \left[S_1\left(k_y + (\Delta_{\text{ACS}} + i)\,\Delta k_y,\, k_{x_0} + s\Delta k_x\right) \right]_{0 \le s < \Delta s}^{1 \le i < R} \\ \vdots \quad\quad \vdots \quad\quad \vdots \\ \left[S_{N_c}\left(k_y + (\Delta_{\text{ACS}} + i)\,\Delta k_y,\, k_{x_0} + s\Delta k_x\right) \right]_{0 \le s < \Delta s}^{1 \le i < R} \end{pmatrix} \in \mathbb{C}^{[(R-1)N_c] \times \Delta s},$$

and finally solve the extended system for W:

$$\underbrace{\left[\hat{B}_{\left(k_{y_0} + i\Delta k_y\right)} \right]_{0 \le i < N_b^{\text{ACS}}}}_{B} = W \cdot \underbrace{\left[\hat{A}_{\left(k_{y_0} + i\Delta k_y\right)} \right]_{0 \le i < N_b^{\text{ACS}}}}_{A}. \tag{4.2}$$

The first column of the considered segment is denoted by k_{x_0} and the step size in PE-direction by Δk_x. To point out the configuration of A and B, it is once again visualized in Figure 4.1.

The same matrix pattern is of course also used for the reconstruction of the skipped lines, once the coefficients are computed. Obviously, N_b neighboring reference points within each channel contribute to the reconstruction of a single missing element.

The normalization of the matrices A and B in Step 2 results in better numerical stability. The corresponding normalization parameter η as well as the regularization parameter χ involved in the computation of λ in Step 4 are user defined and have to be chosen very carefully.

The computation of the unknown fitting coefficients W of each segment is outlined in Algorithm 4.2 (`findWs`).

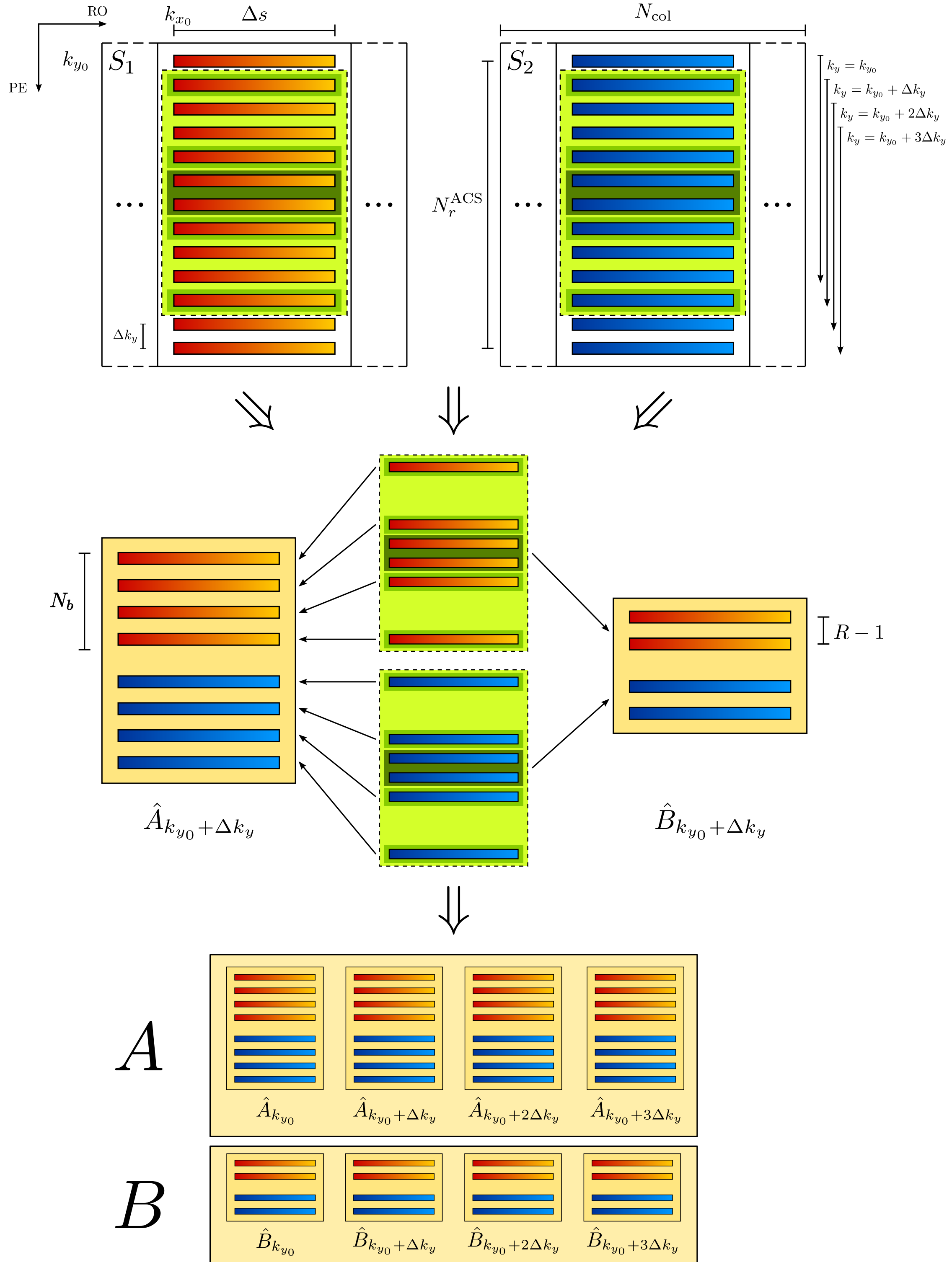


Figure 4.1: Matrix Structure in Basic GRAPPA Autocalibration
for $N_c = 2$, $R = 3$, and $N_b = 4$. The green fitting mask is slided along PE-direction over all ACS lines. Each mask shift generates another part of the final matrices A and B.

Algorithm 4.2: GRAPPA Autocalibration (`findWs`)

```
Input:
  S:  ACS data
  η:  parameter for normalization
  χ:  parameter for regularization
  h_A: height of A (#rows)
```

```
1   // initialization
    A = []; B = []
3
    for each segment do
5     // Steps 1 + 2
        for each ACS block i do
7         A_seg = Â_{k_y0 + iΔk_y}  // using S
          B_seg = B̂_{k_y0 + iΔk_y}  // using S
9
          p = 0
11        for each element e in B_seg do
            p += ||e||_2^2
13        end
          ψ = p^{-η/2}
15
          A_seg = A_seg .* ψ
17        B_seg = B_seg .* ψ

19        A{0,i}_Δs = A_seg // append A_seg to A
          B{0,i}_Δs = B_seg // append B_seg to B
21      end

23    // Step 3
        U = B * A^H
25
      // Step 4
27      V̂  = A * A^H
        t  = Trace(V̂)
29      λ  = χ * t / h_A
        V̂ += λ * E
31      V  = V̂^{-1}

33    // Step 5
        W_seg = U * V
35      W{0,i}_hA = W_seg // append W_seg to W
    end
```

```
Output:
  W: fitting coefficients for all segments
```

4.1.2 Improved Approach

The improved version for the GRAPPA autocalibration does not only use several (K_{PE}) reference points in PE-direction but also a certain number (K_{RO}) of reference points in RO-direction. In other

words, a two-dimensional filter kernel of size $K_{\mathrm{PE}} \times K_{\mathrm{RO}}$ is applied to the input data. This leads to a slightly different matrix configuration for A and B illustrated in Figure 4.2. The filter is initially placed in the "top-left" corner of the ACS data and is then shifted N_{sh} times: $N_{\mathrm{sh}}^{\mathrm{RO}}$ times in RO-direction and $N_{\mathrm{sh}}^{\mathrm{PE}}$ times in PE-direction[1]:

$$\begin{aligned} N_{\mathrm{sh}} &= N_{\mathrm{sh}}^{\mathrm{RO}} N_{\mathrm{sh}}^{\mathrm{PE}} \\ N_{\mathrm{sh}}^{\mathrm{RO}} &= N_{\mathrm{col}} - K_{\mathrm{RO}} + 1 \\ N_{\mathrm{sh}}^{\mathrm{PE}} &= N_{\mathrm{r}}^{\mathrm{ACS}} - K_{\mathrm{PE}} + 1 \,, \end{aligned}$$

where N_{col} denotes the overall number of k-space columns of each coil and $N_{\mathrm{r}}^{\mathrm{ACS}}$ still the number of ACS rows. The formal description of the matrix layout is then given by:

$$\begin{aligned} \widetilde{A}_{k_y} &:= \left[\widetilde{A}_{k_y}^{c}\right]_{0 \le c < N_{\mathrm{sh}}^{\mathrm{RO}}} \in \mathbb{C}^{[K_{\mathrm{PE}} \cdot N_c \cdot K_{\mathrm{RO}}] \times N_{\mathrm{sh}}^{\mathrm{RO}}} \,, \\ \widetilde{B}_{k_y} &:= \left[\widetilde{B}_{k_y}^{c}\right]_{0 \le c < N_{\mathrm{sh}}^{\mathrm{RO}}} \in \mathbb{C}^{[(R-1) N_c] \times N_{\mathrm{sh}}^{\mathrm{RO}}} \,, \\ \widetilde{A}_{k_y}^{c} &:= \left[\begin{pmatrix} \left[S_1\left(k_y + iR\Delta k_y,\, k_{x_0} + s\,\Delta k_x\right)\right]^{0 \le j < K_{\mathrm{RO}}} \\ \vdots \qquad \vdots \qquad \vdots \\ \left[S_{N_c}\left(k_y + iR\Delta k_y,\, (c+j)\,\Delta k_x\right)\right]^{0 \le j < K_{\mathrm{RO}}} \end{pmatrix}\right]^{0 \le i < K_{\mathrm{PE}}} \in \mathbb{C}^{[K_{\mathrm{PE}} \cdot N_c \cdot K_{\mathrm{RO}}] \times 1} \,, \\ \widetilde{B}_{k_y}^{c} &:= \begin{pmatrix} \left[S_1\left(k_y + (\Delta_{\mathrm{PE}} + i)\,\Delta k_y,\, (c + \Delta_{\mathrm{RO}})\,\Delta k_x\right)\right]^{1 \le i < R} \\ \vdots \qquad \vdots \qquad \vdots \\ \left[S_{N_c}\left(k_y + (\Delta_{\mathrm{PE}} + i)\,\Delta k_y,\, (c + \Delta_{\mathrm{RO}})\,\Delta k_x\right)\right]^{1 \le i < R} \end{pmatrix} \in \mathbb{C}^{[(R-1) N_c] \times 1} \,, \\ \Delta_{\mathrm{RO}} &= \left\lfloor \frac{K_{\mathrm{RO}} - 1}{2} \right\rfloor \,, \qquad \Delta_{\mathrm{PE}} = R \cdot \left(\left\lfloor \frac{K_{\mathrm{PE}}}{2} \right\rfloor - 1 \right) \,. \end{aligned} \tag{4.3}$$

The resulting overdetermined linear equation system

$$\underbrace{\left[\widetilde{B}_{k_{y_0} + i\Delta k_y}\right]_{0 \le i < N_{\mathrm{sh}}^{PE}}}_{B} = W \cdot \underbrace{\left[\widetilde{A}_{k_{y_0} + i\Delta k_y}\right]_{0 \le i < N_{\mathrm{sh}}^{PE}}}_{A} \tag{4.4}$$

is finally solved for W.

[1] initial position of the filter kernel included in each case

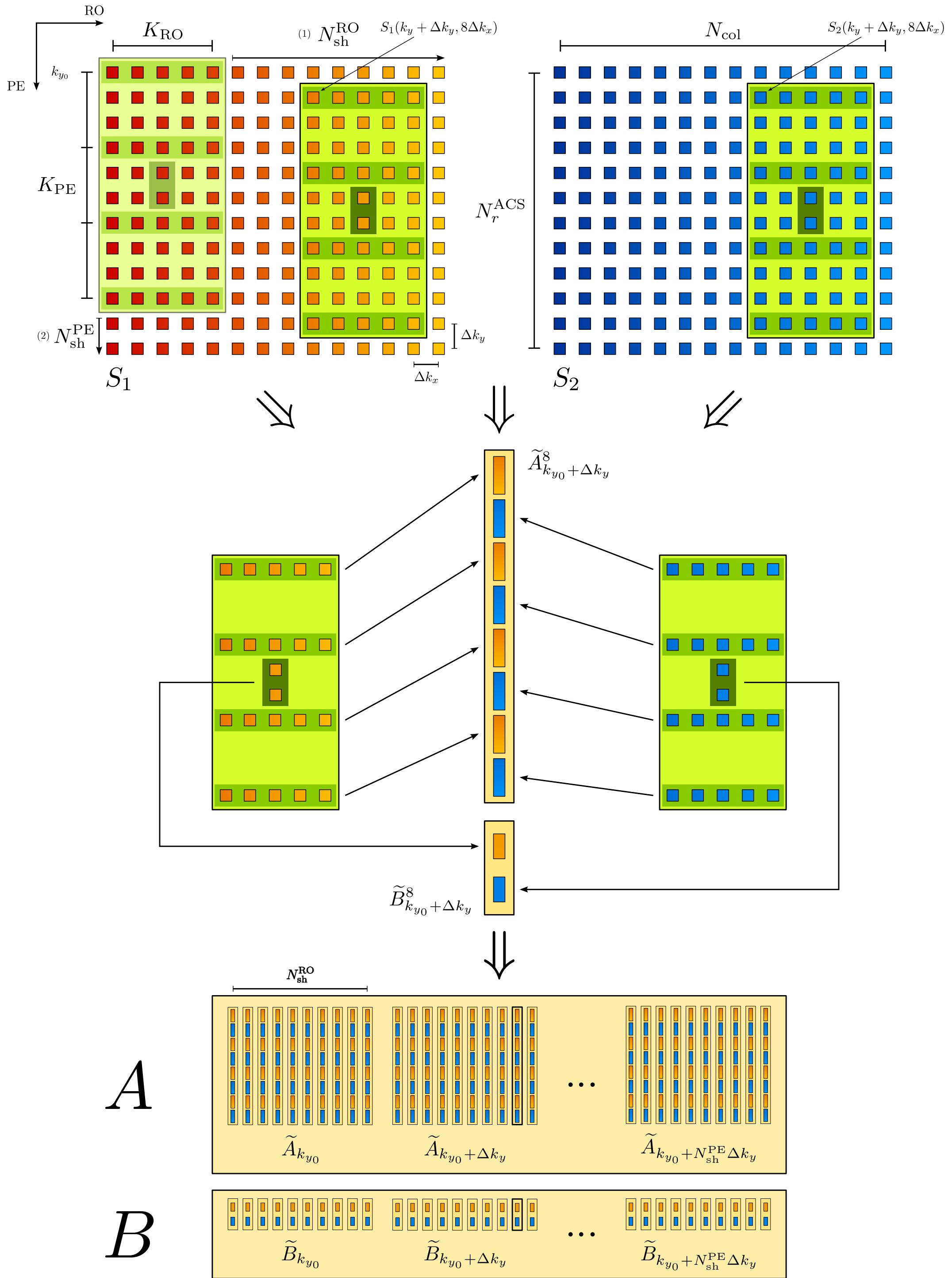


Figure 4.2: Matrix Structure in Improved GRAPPA Autocalibration for $N_c = 2$, $K_{\mathrm{RO}} = 5$, and $K_{\mathrm{PE}} = 4$. The green $K_{\mathrm{PE}} \times K_{\mathrm{RO}}$ filter kernel is shifted along RO - and PE - direction. Each mask position generates another column vector for the matrices A and B.

Besides the different matrix layout, there is an additional *power clipping step* in the improved algorithm before the slightly adjusted normalization in Step 2. The other steps of the improved approach remain the same. The keynote of the clipping step is to exclude a certain amount κ of "interfering" equations from the center of k-space to improve numerical stability. The selection is based on the corresponding "coefficient power". Further details are given in the comprehensive description of `findWsImproved` in Algorithm 4.3:

Algorithm 4.3: Improved GRAPPA Autocalibration (`findWsImproved`)

```
Input:
  S: ACS data
  η: parameter for normalization
  χ: parameter for regularization
  κ: parameter for power clipping
```

```
1   // initialization
2     A = []; B = []
3
4     nD = κ * N_sh              // number of equations to discard
5     list = SortedList(nD)      // sorted list for nD (colIdx, pow) pairs
6
7   // Steps 1 + 2
8     for i = 0 to N_sh^PE-1 do
9       p = 0
10      A_RO = []; B_RO = []
11
12      for j = 0 to N_sh^RO-1 do
13        a_col = Ã_i^j; b_col = B̃_i^j
14
15        pA = ||a_col||_2^2; pB = ||b_col||_2^2
16        p_col = pA * nRowB + pB * nRowA
17        p += p_col
18
19        colIdx = i * N_sh^RO + j
20        list.insertSort((colIdx, p_col))
21
22        A_RO[•,j] = a_col
23        B_RO[•,j] = b_col
24      end
25
26      ψ = p^(-η/2)
27      A_RO = A_RO .* ψ
28      B_RO = B_RO .* ψ
29
30      A{0,i}_{N_sh^RO} = A_RO // append A_RO to A
31      B{0,i}_{N_sh^RO} = B_RO // append B_RO to B
32    end
33
```

```
    // power clipping
35    for i = 0 to nD-1 do
        idx_discard = list[i].colIdx
37      A[•,idx_discard] = 0; B[•,idx_discard] = 0
      end
39
    // Step 3
41    U = B * A^H

43  // Step 4
      M  = A * A^H
45    t  = Trace(M)
      λ  = χ * t / #aRow
47    M += λ * E
      V  = M^-1
49
    // Step 5
51    W = U * V
```

```
Output:
  W: fitting coefficients
```

4.1.3 Computational Costs and Complexity

As we have seen, the two approaches only differ in the first steps. Once, the matrices A and B are filled with accordingly normalized values and clipped, the remaining Steps 3 to 5 are the same. These final steps include the computationally most intensive and hence most time-consuming operations of the whole autocalibration algorithm, i.e. complex-valued *matrix multiplication* and *matrix inversion*. We trace back the complex-valued operations to FP operations to get a reasonable estimation for the total computational costs.

Complex numbers In the following, z_1 and z_2 are assumed to be complex numbers and a,b,c,d to be FP numbers such that

$$
\begin{aligned}
z_1 &= a + b\,i \quad \in \mathbb{C} \\
z_2 &= c + d\,i \quad \in \mathbb{C}\ .
\end{aligned}
$$

Complex addition The addition of z_1 and z_2 requires two FP additions according to

$$z = z_1 + z_2 = (a+c) + (b+d)\,i\ .$$

Complex multiplication The product z of z_1 and z_2 is

$$z = z_1 \cdot z_2 = (ac - bd) + (ad + bc)\, i \,,$$

and takes four FP multiplications and two FP additions[2].

Matrix Multiplication The multiplication of two matrices $X \in \mathbb{C}^{r\times s}$ and $Y \in \mathbb{C}^{s\times t}$ is defined as

$$\begin{aligned} Z &= X \cdot Y \\ z_{i,j} &= \sum_{k=0}^{s} x_{i,s} \cdot y_{s,j} \,, \end{aligned}$$

$$\text{where} \quad Z = \left[z_{i,j}\right]_{0\le j<t}^{0\le i<r} \in \mathbb{C}^{r\times t} \,, \quad X = \left[x_{i,j}\right]_{0\le j<s}^{0\le i<r} \in \mathbb{C}^{r\times s} \,, \quad Y = \left[y_{i,j}\right]_{0\le j<t}^{0\le i<s} \in \mathbb{C}^{s\times t} \,.$$

The computation of each element of Z hence requires s complex multiplications and s complex additions[3]. As matrix Z consists of $r \cdot t$ elements, the overall computational effort adds up to $r \cdot t \cdot s$ complex multiplications and complex additions respectively.
If we neglect the difference between FP multiplication and FP addition and count basic FP operations provided by a processor we end up with

$$\Sigma^{\text{mmul}} = \vartheta \cdot (r \cdot s \cdot t) \,, \quad \vartheta \in \mathbb{N} \tag{4.5}$$

FP operations for the multiplication of A and B. The constant ϑ depends on the supported instruction set of the processing unit[4]. The simplest case is a one-to-one mapping, which results in $\vartheta = 2 + (2 + 4) = 8$. If we assume the processor to feature sophisticated operations such as *Multiply-Add (madd)* we can achieve $\vartheta = 5$ or even $\vartheta = 4$ depending on the handling of the subtraction within the complex multiplication.

Now we get back to the actual problem and include these preliminary considerations concerning complex-valued matrix multiplication. First of all, we have to look back on the occurring matrices and

[2] addition and subtraction are equated

[3] Strictly speaking, there are only $s - 1$ additions as the first one is for free. This detail is neglected to keep things as simple as possible.

[4] and the compiler's smartness

	findWs		findWsImproved	
	# rows	# columns	# rows	# columns
A	$N_b \cdot N_c$	$\Delta s \cdot N_b^{\text{ACS}}$	$K_{\text{PE}} \cdot N_c \cdot K_{\text{RO}}$	N_{sh}
B	$(R-1)\,N_c$		$(R-1)\,N_c$	

Table 4.1: Matrix Dimensions in GRAPPA Autocalibration

step	operation	findWs[a]	findWsImproved
3	$U = B \cdot A^H$	$(R-1)\,N_b\,(N_c)^2\,N_b^{\text{ACS}} N_{\text{col}}$	$(R-1)\,K_{\text{PE}} K_{\text{RO}}\,(N_c)^2\,N_{\text{sh}}$
4[b]	$\hat{V} = A \cdot A^H$	$(N_b)^2\,(N_c)^2\,N_b^{\text{ACS}} N_{\text{col}}$	$(K_{\text{PE}})^2\,(K_{\text{RO}})^2\,(N_c)^2\,N_{\text{sh}}$
5	$W = U \cdot V$	$(R-1)\,(N_b)^2\,(N_c)^3$	$(R-1)\,(K_{\text{PE}})^2\,(K_{\text{RO}})^2\,(N_c)^3$

[a] Step 3 and 4 are repeated for all $N_{\text{col}}/\Delta s$ segments

[b] regularization and inversion unconsidered

Table 4.2: Computational Complexity of GRAPPA Autocalibration
The table lists the total number of FP operations as multiple of ϑ for matrix multiplications in the GRAPPA autocalibration.

their dimensions shown in Table 4.1 (see also Equations 4.2, 4.3, 4.4). Table 4.2 lists the computational costs of Steps 3 to 5 according to Equation 4.5 and Table 4.1.

We want to point out that the number of channels N_c is crucial as it is involved cubically and hence a key factor. Even though arrays of 64 and 128 coils are already available for exploratory MR systems, their practical application has been all but possible up to now due to unacceptable reconstruction times. Current MR systems usually work with up to 32 input channels. However, the whole set of coils is solely involved in the autocalibration step, whereas the reconstruction stage only uses a fraction of them as output channel. The *channel reduction* reduces the number of rows in B and speeds up the whole image reconstruction as there are much fewer lines to reconstruct and to transform back into image space.

4.2 Matrix Multiplication

As the complex-valued matrix multiplication is the most time-consuming operation in the GRAPPA autocalibration, this section addresses this problem and presents different GPU approaches.
We consider the multiplication of two complex-valued matrices

$$\begin{aligned} X \in \mathbb{C}^{r\times s}, \quad Y \in \mathbb{C}^{s\times t} \quad , \qquad & \text{where} \quad \frac{r}{\Delta b_x}, \frac{s}{\Delta b_k}, \frac{t}{\Delta b_y} \in \mathbb{N} \\ Z \;=\; (X \cdot Y) \in \mathbb{C}^{r\times t} \qquad & \Delta b_x, \Delta b_k, \Delta b_y \in \mathbb{N} \; . \end{aligned} \tag{4.6}$$

The additional demands on matrix dimensions do not profoundly restrict the general applicability as each matrix can be extended with extra rows and columns filled with zeros (*padding*) to fulfill the conditions. We assume both (padded) matrices X and Y to be initially stored in global memory. Moreover, X, Y, and Z are considered as synonyms for the matrices itself on the one hand and the corresponding locations in global memory on the other hand. At this point, we would like to draw your attention to Section A.2 for further details on matrix storage and access.

4.2.1 Basic Approach

First of all, we explain a straightforward approach also described in the CUDA Programming Guide [61] and implemented as a showcase in the *CUDA 1.1 Software Development Kit (SDK)* [63].

The "blocked" execution model of CUDA suggests to partition output data (matrix Z) into tiles of size $\Delta b_x \times \Delta b_y$ that are mapped to thread blocks. Each thread computes a single tile entry as the inner product of a single row of X and a single column of Y. In doing so, threads in the same row of a block may share their X-row, threads in the same block-column their Y-column, respectively. We combine row- and column-sharing to avoid multiple redundant accesses to global memory by initially loading parts of the required data into shared memory (see also Section 3.3.7). In general it is impossible to load all Δb_x X-rows and Δb_y Y-columns at once as shared memory is a quite limited resource. The computation of the inner product is hence split up into consecutively processed segments of size Δb_k as shown in Figure 4.3. This segmentation results in a reduced and particularly feasible amount of shared memory required all at once, i.e. $\Delta b_k\,(\Delta b_x + \Delta b_y)$ elements instead of $s\,(\Delta b_x + \Delta b_y)$.

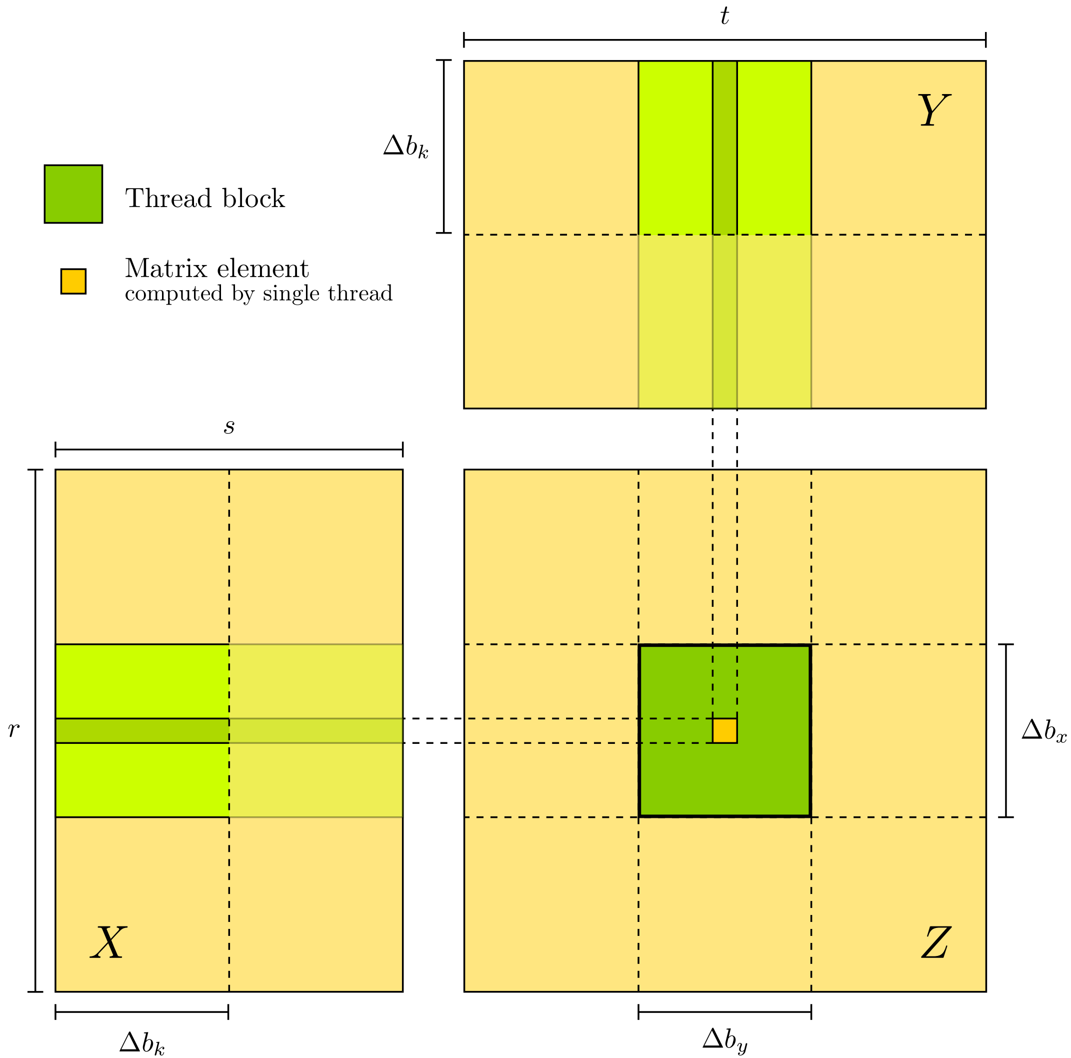


Figure 4.3: Basic Scheme for Matrix Multiplication $Z = XY$
Each thread block computes one submatrix of Z. A single entry of each submatrix corresponds to one thread that iteratively computes the inner product in groups of size Δb_k.

The interim result of the segmented inner product is stored per thread in a local accumulator that is finally written to Z in global memory. Moreover, we have chosen square blocks for a start in Algorithm 4.4, i.e. $\Delta b := \Delta b_x = \Delta b_y = \Delta b_k$.

Algorithm 4.4: Basic Approach for Matrix Multiplication (`mmul1`)

```
Input:
  X,Y: input matrices
```

```
   // initialization
2    shared complex shX[Δb][Δb]
     shared complex shY[Δb][Δb]
4    complex accu = (0,0)

6  for each segment i do
     // load required data into shared memory
8      shX[ty,tx] = X{by,i}[ty,tx]
       shY[ty,tx] = Y{i,bx}[ty,tx]
10     synchronize

12   // compute partial inner product
       for k = 0 to Δb-1 do
14       accu += shX[ty,k] * shY[k,tx]
       end
16     synchronize
   end
18
   Z{by,bx}_Δb[ty,tx] = accu // write accu to Z
```

```
Output:
  Z = X * Y
```

The **`synchronize`** operation at line 10 assures that all required input elements are loaded into shared memory and can be processed. In the same way, all threads have to be done computing the partial inner product before the shared memory is overwritten (see line 16).

The basic approach uses 16 registers and buffers $2\Delta b^2$ complex numbers in shared memory. A single complex number consists of two FP numbers of 4 Byte (B) each. The amount of used shared memory therefore amounts to $16\Delta b^2$ Byte. The CUDA Occupancy Calculator [58] might help to "optimize" the choice of the block size Δb in terms of MP warp occupancy. As already explained, though, occupancy is only one factor besides many others and maximizing it does not always result in an optimal overall performance! For this kernel $\Delta b = 16$ is a good choice, i.e. there are $\Delta b^2 = 256$ threads per block and approximately 4 kB of shared memory per block, which results in an occupancy of 67%.

4.2.2 Improved Kernels

Our first attempt was to achieve higher arithmetic intensity by reusing shared data within each thread and hence avoid redundant memory accesses. The thread blocks remain at the same size $\Delta b \times \Delta b$ but the output matrix Z is now segmented into square tiles with a side length of $2\Delta b$ as illustrated in Figure 4.4.

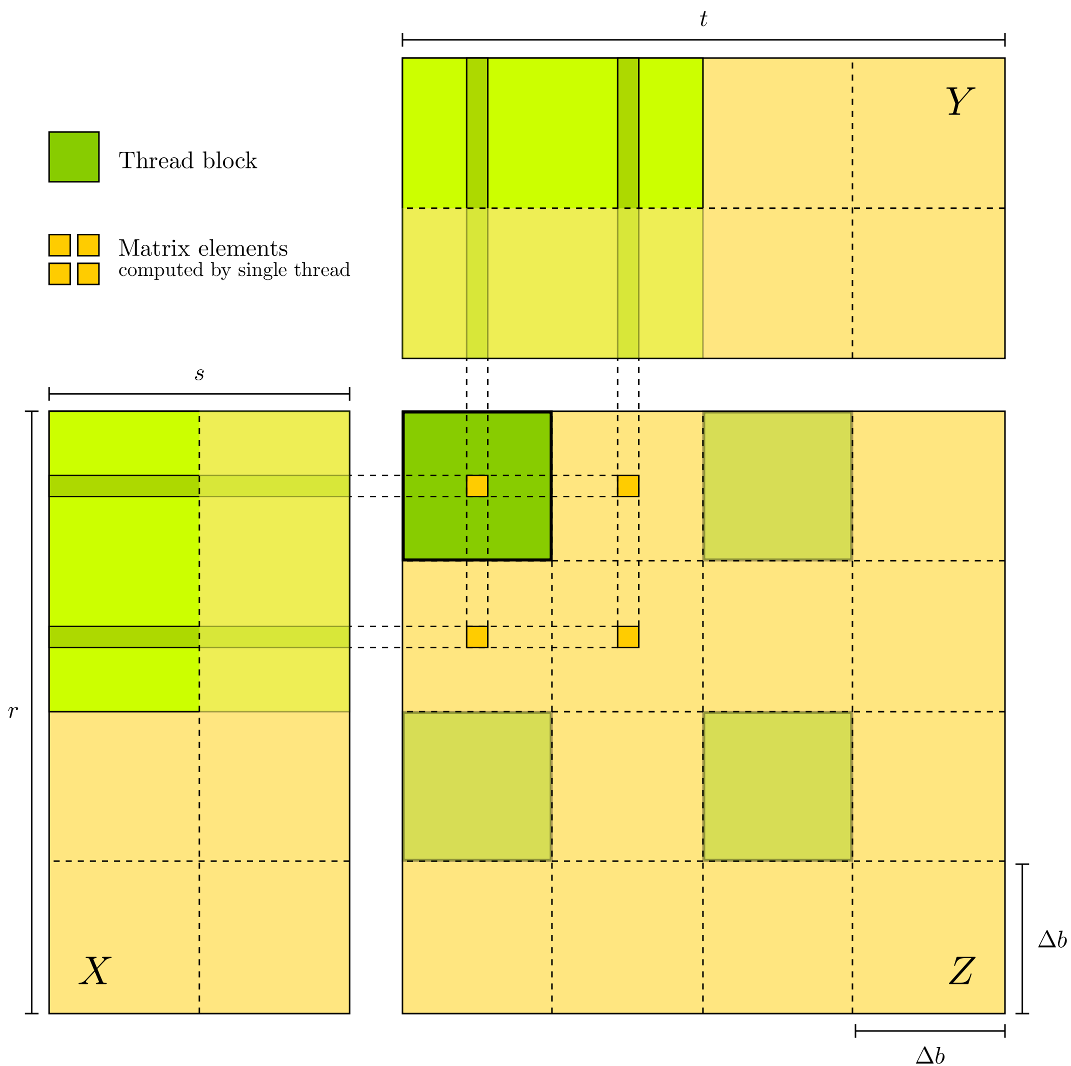


Figure 4.4: Adjusted Scheme for Matrix Multiplication $Z = XY$
Each thread block computes one Z-tile of size $2\Delta b \times 2\Delta b$. A single thread iteratively computes four elements.

Each thread loads two elements of X and Y into shared memory in each iteration of the inner product and computes four elements of Z as there are four times more output entries in a Z-block than threads in this case. The according kernel is implemented as described in Algorithm 4.5.

Algorithm 4.5: Improved Approach for Matrix Multiplication (`mmul2`)

```
Input:
  X,Y: input matrices
```

```
  // initialization
2   shared complex shX[2Δb][ Δb]
    shared complex shY[ Δb][2Δb]
4   complex accu[4] = {(0,0),(0,0),(0,0),(0,0)}

6 for each segment i do
    // load required data into shared memory
8     shX[ty,tx]    = X{by,i}_2Δb[ty,tx]
      shX[ty+Δb,tx] = X{by,i}_2Δb[ty+Δb,tx]
10    shY[ty,tx]    = Y{i,bx}_2Δb[ty,tx]
      shY[ty,tx+Δb] = Y{i,bx}_2Δb[ty,tx+Δb]
12    synchronize

14  // compute partial inner product
      for k = 0 to Δb-1 do
16      accu[0] += shX[ty,k]    * shY[k,tx]
        accu[1] += shX[ty+Δb,k] * shY[k,tx]
18      accu[2] += shX[ty,k]    * shY[k,tx+Δb]
        accu[3] += shX[ty+Δb,k] * shY[k,tx+Δb]
20    end
      synchronize
22 end

24 // write results to Z
    Z{by,bx}_Δb[ty,tx]          = accu[0]
26  Z{by,bx+Δb}_Δb[ty,tx]       = accu[2]
    Z{by+Δb,bx}_Δb[ty,tx]       = accu[1]
28  Z{by+Δb,bx}_Δb[ty,tx+Δb]    = accu[3]
```

```
Output:
  Z = X * Y
```

As there are many more indices to compute, this kernel needs 27 registers per thread, which is quite much and lowers the number of concurrently processed blocks per MP according to Equation 3.1. The amount of shared memory is approximately twice as high compared to the basic approach, i.e. 8 kB for $\Delta b = 16$. All in all, the (reasonably) increased utilization of shared memory and registers scales down the occupancy to 33%. Nevertheless, the overall performance turns out to be much better than

mmul1 (see Chapter 5.1). As already mentioned, high occupancy is desirable but no absolute must! In this case, benefits due to enhanced data sharing outweigh the rather poor occupancy.

In a next step, we contemplate algorithm mmul2 with respect to shared memory access and bank conflicts in particular. For this purpose, we have to go into more detail at first.
Shared memory is organized in 16 banks of 32 bit bandwidth per two clock cycles. Successive 32-bit words in shared memory space are assigned to successive banks that can be accessed simultaneously by a *half-warp*[5] as long as threads access distinct banks. Otherwise, access operations have to be serialized in general except for a few cases [61]. One of those exceptions worth mentioning is the "one-for-all" situation, i.e. all threads of a half-warp access exactly the same address in shared memory. This access is not decomposed into 16 separate conflict-free requests but performed in parallel. Moreover, it is important to know that bank conflicts may occur only for threads within the same half-warp.

Now we get back to our actual algorithm and concentrate on the computation of the inner product in lines 16 to 19 of Algorithm 4.5. Each thread computes four elements located in distinct output blocks of size $\Delta b \times \Delta b$ (see Figure 4.4). It is adequate to examine only the first computation (line 16) since the situation is the same for the others.
A single block row r_b consists of $\Delta b = 16$ threads with fixed ty = r_b and tx $\in \{0, \cdots, 15\}$. In other words, each block row forms a half-warp and two consecutive rows a warp.
Consider the situation in line 16 for the first half warp:

```
for k = 0 to Δb-1 do
  accu[0] += shX[ty,k] * shY[k,tx]
  // [...]
end
```

The access to shX only depends on ty and k which are equal for all threads. The whole half-warp, thus, accesses the same address in shared memory, which is the "one-for-all" situation mentioned above. By contrast, the read operation on shY depends on varying tx. Complex numbers are linearly aligned in shared memory and span two consecutive banks each (64 bit). Consequently, there are two 32-bit read operations involved in a single 64-bit request accessing two consecutive banks in shared

[5] first or second 16 threads of a warp

memory. The resulting bank access pattern for shY of a half-warp is listed in Table 4.3.

	#op	tx 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bank	1	00	02	04	06	08	10	12	14	00	02	04	06	08	10	12	14
	2	01	03	05	07	09	11	13	15	01	03	05	07	09	11	13	15

Table 4.3: Access Pattern for shY with Bank Conflicts

It becomes clear that in the first 32-bit operation only banks with even indices are involved and odd indices in the second one, respectively, resulting in half memory throughput.

Luckily, we can easily resolve these 2-way bank-conflicts by padding shY with 32 bits every eight complex numbers. In this way, we obtain the conflict-free access pattern for shY sketched in Table 4.4.

	#op	tx 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bank	1	00	02	04	06	08	10	12	14	01	03	05	07	09	11	13	15
	2	01	03	05	07	09	11	13	15	02	04	06	08	10	12	14	00

Table 4.4: Conflict-Free shY Access Pattern for ty $\in \{0,4,8,12\}$

The shY access pattern now also depends on ty because of the padding bytes. To make Table 4.4 universally valid, each bank index table entry i has to be considered as $i' = (i + 4 \cdot \mathrm{ty})$ *mod* 16 as a single row of the unpadded shY has $2\Delta b$ elements and therefore has to be padded with four 32-bit blocks in order to avoid bank conflicts for the whole computation of the inner product.

We have to adjust only few lines of Algorithm 4.5 to incorporate conflict-free access to shared memory. The additional four 32-bit segments for shY correspond to two additional complex numbers per row. To simplify matters, we assume shY[ty,tx]$_{\text{off}}$ to denote the element of shY in row ty and column tx with respect to 32-bit offset off. In the practical implementation this can easily be done using pointer arithmetic.

Algorithm 4.6: Improved Approach for Matrix Multiplication (`mmul3`)

```
Input:
  X,Y: input matrices
```

```
   // initialization
 2   shared complex shX[2Δb][Δb]
     shared complex shY[ Δb][2(Δb+1)]
 4   complex accu[4] = {(0,0),(0,0),(0,0),(0,0)}

 6   o1 = tx/8; o2 = (tx+Δb)/8 // offsets for padding

 8 for each segment i do
     // load required data into shared memory
10     shX[ty,tx]       = X{by,i}_2Δb[ty,tx]
       shX[ty+Δb,tx]    = X{by,i}_2Δb[ty+Δb,tx]
12     shY[ty,tx]_o1    = Y{i,bx}_2Δb[ty,tx]
       shY[ty,tx+Δb]_o2 = Y{i,bx}_2Δb[ty,tx+Δb]
14     synchronize

16   // compute partial inner product
       for k = 0 to Δb-1 do
18       accu[0] += shX[ty,k]    * shY[k,tx]_o1
         accu[1] += shX[ty+Δb,k] * shY[k,tx]_o1
20       accu[2] += shX[ty,k]    * shY[k,tx+Δb]_o2
         accu[3] += shX[ty+Δb,k] * shY[k,tx+Δb]_o2
22     end
       synchronize
24 end

26 // write results to Z
     Z{by,bx}_Δb[ty,tx]        = accu[0]
28   Z{by,bx+Δb}_Δb[ty,tx]     = accu[2]
     Z{by+Δb,bx}_Δb[ty,tx]     = accu[1]
30   Z{by+Δb,bx}_Δb[ty,tx+Δb]  = accu[3]
```

```
Output:
  Z = X * Y
```

Please note, that (integer) division is particularly costly and can be replaced by a bitwise shift operation in line 6. The number of registers remains the same compared to algorithm `mmul2` and the amount of utilized shared memory hardly increased without any influence on the occupancy.

Maybe you wonder why we have not applied the padding technique to `shX` but accept bank conflicts in lines 10 and 11. Basically, there are two reasons for this decision. On the one hand, bank conflicts while accessing global memory are all but negligible because of high latency. On the other hand, we would have to compute a new offset for each loop iteration in lines 18 to 21 based on the `shX` column index `k` in contrast to `o1` and `o2` that have to be computed only once per thread. Therefore, padding

shX is a change for the worse, whereas resolving bank conflicts for shY results in a considerable gain of performance.

Finally, we tried some non-square block-layouts with unexpected effects. For this purpose, we have to distinguish between Δb_x, Δb_y and Δb_k introduced in Equation 4.6. Moreover, Δb_x is chosen as a multiple of a small integer μ. The output matrix Z is decomposed into tiles of size $\Delta b_x \times \Delta b_y$ that are assigned to thread blocks of size $\Delta b_x / \mu \times \Delta b_y$. Each thread computes μ elements in a single column of a tile but the basic concept remains the same. The whole kernel is specified in Algorithm 4.7.

Algorithm 4.7: Improved Approach for Matrix Multiplication (mmul4)

```
Input:
  X,Y:  input matrices
  Δb_k: block size for iterated inner product (≤ Δb_y)
  s:    width of X and height of Y
```

```
   // initialization
2    shared complex shX[Δb_x][Δb_k]
     complex accu[μ];
4
     for i = 0 to μ-1 do
6      accu[i] = (0,0);
     end
8
   for i = 0 to s/Δb_k-1 do
10   // load required data into shared memory
       if tx < Δb_k then
12       for j = 0 to μ-1 do
           shX[μ*ty+j,tx] = X{by,i}^{Δb_x}_{Δb_k}[μ*ty+j,tx]
14       end
       end
16     synchronize

18   // compute partial inner product
       for k = 0 to Δb_k-1 do
20       for j = 0 to μ-1 do
           accu[j] += shX[ty*μ+j,k] * Y{i,bx}^{Δb_k}_{Δb_y}[k,tx]
22       end
       end
24     synchronize
   end
26
   // write results to Z
28   for i = 0 to μ-1 do
       Z{by,bx}^{Δb_x}_{Δb_y}[ty+i,tx] = accu[i]
30   end
```

```
Output:
  Z = X * Y
```

We tried several configurations for Δb_x, Δb_y, Δb_k, and μ with respect to some restrictions:

- $\frac{\Delta b_x \Delta b_y}{\mu} \leq 512$ (maximum number of threads per block)
- $\Delta b_k \leq \Delta b_y$ (enough threads to load `shX`, see line 13)
- $\Delta b_x \Delta b_k \ll 2^{11}$ (limited amount of shared memory)

The preparatory loading of Y into shared memory (`shY`) is omitted as narrow, wide blocks turned out to perform best particularly for large input matrices. Otherwise, $\Delta b_x/\mu \times \Delta b_y$ threads in a block would have to load `shX` and `shY` of size $\Delta b_x \times \Delta b_k$ respectively $\Delta b_k \times \Delta b_y$. This task requires additional index computations resulting in an increased number of registers per thread. In combination with the increased amount of shared memory, this causes poor occupancy.
We finally decided for $\Delta b_x = \mu = 4$, and $\Delta b_y = \Delta b_k = 64$. In other words, a block consists of one single row of 64 threads for this configuration, which is illustrated in Figure 4.5. Therefore, the use of `shY` is redundant anyway as each thread works on different Y columns. Moreover, the increased block width benefits the utilization of `shX`. All in all, we accomplished our final kernel using 24 registers per thread and approximately 2 kB of shared memory.

Finally, some additional remarks about the for-loops in Algorithm 4.7. The NVIDIA compiler nvcc supports simple loop unrolling since version 1.1 [63]. Needless to say, that the compiler can unroll only loops with a trip count known at compile time. The parameters Δb_x, Δb_y, Δb_k, and μ are incorporated as templates so that the for-loops in lines 5, 12, 20, and 28 can be unrolled. Using previous versions of nvcc, however, loops have to be unrolled manually, which is quite tedious.

4.2.3 CUBLAS

For the multiplication of complex-valued matrices, the higher-level library CUBLAS provides the *complex general matrix multiplication* (`cublasCgemm()`) as the only single-precision complex BLAS3 function [56]. It takes two complex matrices X and Y and two complex numbers c_1 and c_2 as input

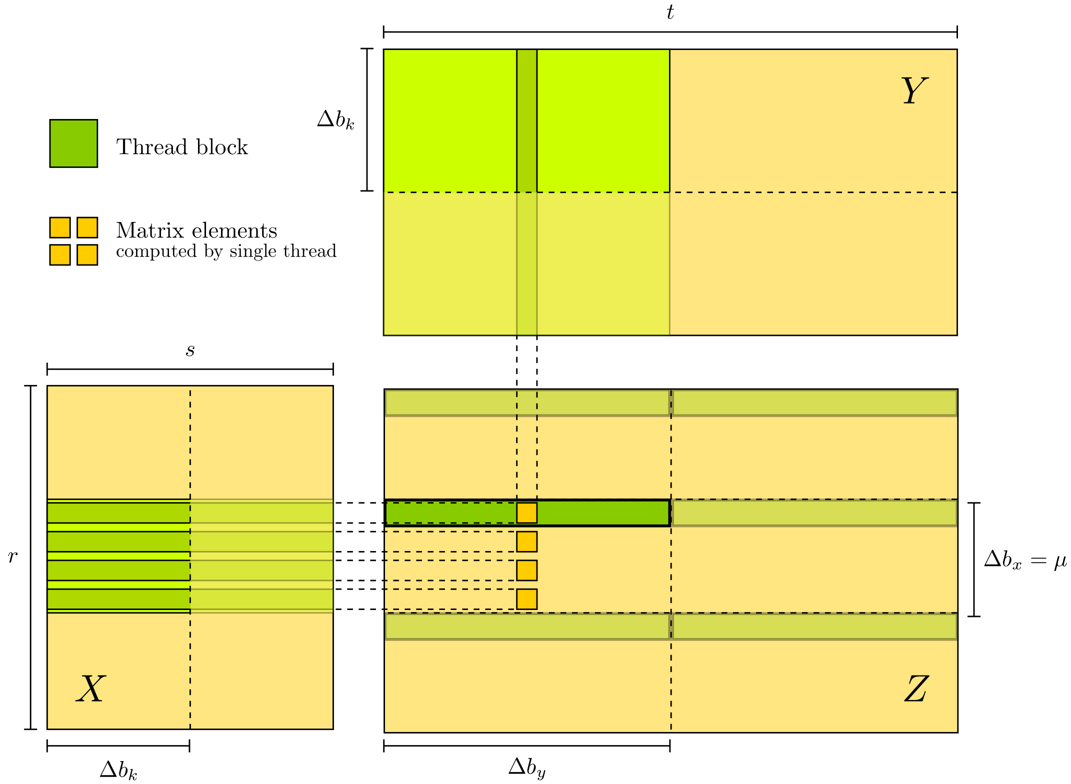


Figure 4.5: Optimized Scheme for Matrix Multiplication $Z = XY$
For $\Delta b_x = \mu$, the blocks consist of Δb_y threads each computing μ elements of the corresponding $\Delta b_x \times \Delta b_y$ submatrix of Z.

and performs the following general operation:

$$
\begin{aligned}
Z &= c_1 \cdot \mathrm{op}_X(X) \cdot \mathrm{op}_Y(Y) + c_2 \cdot Z\,, \\
\text{where} \quad & c_1, c_2 \in \mathbb{C} \ \text{ and } \ \mathrm{op}(\bullet) \in \left\{ \bullet, \bullet^T, \bullet^H \right\} \ .
\end{aligned}
$$

It is important to know that CUBLAS expects matrices to be stored in *column-major order* (see Section A.2). This has to be taken into consideration for the function call. If we want to compute for instance $Z = X \cdot Y$ for row-major aligned matrices this translates to

$$
\widetilde{Z} = (X \cdot Y)^T = Y^T \cdot X^T = 1 \cdot \widetilde{Y} \cdot \widetilde{X} + 0 \cdot \widetilde{Z}
$$

for `cublasCgemm()`, where $\widetilde{X}$, $\widetilde{Y}$, and $\widetilde{Z}$ denote the matrix data X, Y, and Z reinterpreted as column-major aligned by CUBLAS.

According to NVIDIA developers, best performance is achieved in CUDA 1.1 on currently shipping hardware if all matrix dimensions are multiples of 16 and the matrices' addresses are aligned to 128-byte boundary. Moreover, NVIDIA published the source code of the CUBLAS implementation in February 2008 [62]. We already concentrated on incorporating our implementations into the GRAPPA reconstruction pipeline of Siemens Medical Solutions at that time. However, it is definitely worth contemplating kernel details particularly with regard to optimizations for small matrices.

4.2.4 Special case

Up to now, we only considered the general multiplication of two complex-valued matrices. If we take a closer look at the matrix multiplications in Listing 4.1 for the GRAPPA autocalibration we come across the special case in Step 4:

$$\hat{V} = A \cdot A^H \ .$$

The resulting matrix $\hat{V}$ is a *self-adjoint* matrix also known as *Hermitian matrix*, i.e. $\hat{V} = \hat{V}^H$. We can benefit from this "Hermitian symmetry" as the lower left part of the matrix can easily be deduced from the upper right one (or vice versa) by computing the complex conjugate of each element, i.e. flip the sign of the *imaginary part*.

We have tried several different possibilities to map this problem to CUDA. The final result in Algorithm 4.8 is substantially based on `mmul4` but with a quadratic block layout with $\Delta b := \Delta b_x = \Delta b_y = \Delta b_k$. The regular grid is retained for the kernel execution and covers the whole output matrix Z to keep index calculations based on block indices within threads as simple as possible. Blocks of the lower left grid part (`bx` $<$ `by`) immediately return without doing anything. Blocks on the grid diagonal (`bx` = `by`) proceed as in `mmul4`, whereas the remaining ones of the (strict) upper right part (`bx` $>$ `by`) imply duplicate writes to global memory for each computed element with respect to Hermetian symmetry. It is, in fact, even possible to use the symmetry property within diagonal blocks. However, the resulting performance slightly decreases due to more complex branching.

Moreover, it is beneficial for the computation of $Z = A \cdot A^H$ to store A^T separately in advance. We will see that A^T is needed for the preceding normalization stage, anyway. Using the transposed matrix as additional input parameter for the kernel allows for completely coalesced memory access in lines 17 and 25. The computation of the inner product requires a slightly modified multiplication operator in line 25 to incorporate the conjugate complex part of A^H:

$$z_1 \otimes z_2 := z_1 \cdot z_2^* , \quad z_1, z_2 \in \mathbb{C} ,$$

where z_2^* denotes the conjugate complex of z_2.

Algorithm 4.8: Improved Matrix Multiplication $Z = X \cdot X^H$ (`mmul5`)

```
Input:
  X:  input matrix
  XT: input matrix (X=X^T)
  Δb: block size
  s:  width of X
```

```
1   if bx < by then
2     return
3   end
4
5   // initialization
6     shared complex shX[Δb][Δb]
7     complex accu[μ];
8
9     for i = 0 to μ-1 do
10      accu[i] = (0,0);
11    end
12
13  for i = 0 to s/Δb-1 do
14    // load required data into shared memory
15      if tx < Δb then
16        for j = 0 to μ-1 do
17          shX[μ*ty+j,tx] = X{by,i}_Δb[μ*ty+j,tx]
18        end
19      end
20      synchronize
21
22    // compute partial inner product
23      for k = 0 to Δb-1 do
24        for j = 0 to μ-1 do
25          accu[j] += shX[ty*μ+j,k] ⊗ XT{i,bx}_Δb[k,tx]
26        end
27      end
28      synchronize
29  end
```

```
31      // write results to Z
          // fill upper right part
33          for i = 0 to μ-1 do
              Z{by,bx}Δb[ty,tx] = accu[i]
35          end

37        // fill lower left part w.r.t. Hermitian symmetry
            if bx > by then
39            for i = 0 to μ-1 do
                Z{bx,by}Δb[tx,ty] = conjComplex(accu[i])
41            end
            end
```

```
Output:
  Z = X * X^H
```

Roughly speaking, this approach halves the number of operations and execution time as each result is used twice. Unfortunately, there is no possibility to take advantage of the special matrix structure using CUBLAS.

It is even possible to save the write operations to global memory in line 40 if the inversion algorithm inherently makes use of the Hermitian symmetry to compute V and requires only the upper triangular submatrix of the regularized matrix AA^H, e.g. Cholesky Decomposition (see section 5.2).

4.3 Initialization and Normalization

The first steps in the GRAPPA autocalibration stage are the initialization and normalization of A and B according to Listing 4.1 on page 47. In the following, we stick to the basic reconstruction approach (`findWs`) described in Algorithm 4.2. Nevertheless, the findings of the following analysis can easily be applied to `findWsImproved` in a similar way.

4.3.1 Initialization

During initialization, the matrices A and B are filled with proper ACS elements following the pattern of Equation 4.2. Initializing matrices on device is worthwhile simply for the reason that the amount of data to be transferred from host to device is minimized to ACS data[6]. As there are absolutely no arithmetic operations on the input data, there is no possibility to hide high latency access to global memory and the kernels are *band-limited*.

[6]Matrices A and B contain redundant information.

One of the most challenging tasks is to choose a proper thread block layout based on GRAPPA parameters with respect to performance issues and CUDA limitations.
We have chosen the grid layout as $N_b^{\text{ACS}} \times N_c$ with Δs threads per block for both kernels. A single thread (`tx`,`ty`) in block (`bx`,`by`) initializes N_b elements of A and $R-1$ entries of B as listed in Table 4.5 with respect to Equation 4.1. After all blocks have finished, A and B of Equation 4.2 are initialized for a single segment starting at ACS column k_{x_0}.

Kernel	Block (`bx`,`by`)	Thread (`tx`,`ty`)
init A	$\hat{A}_{k_{y_0}}\{\text{by},0\}^{\Delta s}$	$\left[S_{\text{by}}(k_{y_0}+\text{bx}+b\,R\,\Delta k_y,\, k_{x_0}+\text{tx}\,\Delta k_x)\right]^{0\leq b<N_b}$
init B	$\hat{B}_{k_{y_0}}\{\text{by},0\}^{\Delta s}$	$\left[S_{\text{by}}(k_{y_0}+\text{bx}+(\Delta_{\text{ACS}}+i)\,\Delta k_y,\, k_{x_0}+\text{tx}\,\Delta k_x)\right]^{1\leq i<R}$

Table 4.5: Dispatching for Initialization of A and B

The regular structure of both matrices actually allows for loading ACS elements only once and writing them several times to their destination addresses in A and B. On the one hand this technique saves global memory loads, on the other hand this requires complex index calculations and uncoalesced global memory writes, which finally results in poor performance. Another approach stores ACS data in a texture in order to benefit from texture cache effects. Unfortunately, there is a lack of locality in the access pattern so that the cache effect does not become operative.

4.3.2 Normalization

Normalizing A and B requires the following measure p_i for all submatrices of B corresponding to ACS segments i according to line 12 in Algorithm 4.2:

$$p_i \quad = \sum_{e\in B\{0,i\}_{\Delta s}} \|e\|_2^2\,, \quad \forall\, 0 \leq i < \frac{N_{\text{col}}}{\Delta s}\,, \tag{4.7}$$

where the sum runs over all complex-valued elements of $B\{0,i\}_{\Delta s}$.
This kind of operation is also known as *reduction* since all elements (of the submatrix) are processed by the same operator (squared norm) and then reduced (summed up) resulting in a single complex number. In the CPU-based version of `findWs` the reduction can easily be done during the initialization

of the submatrices. In the corresponding kernels, however, this is not as simple due to *read-after-write hazards* and the SPMD architecture. We decided to keep the initialization and the reduction separate even if it is possible to perform parts of the reduction during the initialization[7].

The highly optimized kernel `reduce5()` developed by Harris [25] provided a basis for the reduction problem. It is part of the sample projects in the CUDA 1.1 SDK [63]. The kernel expects an array of integers as input and computes the element-wise sum. The number of integers is assumed to be a power of two[8] since a tree-based approach is used within the blocks. Harris applied sophisticated optimization techniques such as loop unrolling, template programming, and bank conflict avoidance with considerable results. For further particulars and a detailed kernel analysis, please refer to [25]. We merely adjusted the kernel to handle complex numbers and to compute the sum of squared norms for all submatrices of B according to Equation 4.7.

Since the reduction kernel works on a linear array, matrix B has to be *transposed* and possibly padded to a power of two in advance so that the submatrices $B\{0,i\}_{\Delta s}$ of Equation 4.7 are linearly aligned in global memory. We decompose B into tiles of $\Delta b \times \Delta b$ elements ($\Delta b = 16$) for Algorithm 4.9. Each tile is transposed by a thread block of the same size. It is important to take account of coalesced read and write access to global memory as the kernel is memory-bound since latency can not be hidden because of the very poor arithmetic intensity. Therefore, each half-warp writes a single tile row into shared memory, at first. After synchronization a single *column* of the buffered tile in shared memory is written to the corresponding row of the destination tile of B^T. The additional column of `shX` in line 2 avoids bank conflicts during the column-wise readout in line 9.

Once the partial sums p_i of Equation 4.7 are computed, the elements of A and B have to be multiplied with the corresponding normalization coefficients $\psi_i = p_i^{\eta/2}$. Strictly speaking, we normalize the transposed matrices A^T and B^T for reasons of coalesced memory access using Algorithm 4.10 with a $(N_b N_c) \times (N_{\text{col}}/\Delta s)$ grid for A^T and a $(R-1)N_c \times (N_{\text{col}}/\Delta s)$ grid for B^T, respectively, with Δs threads per block.

[7] Each initialization thread could compute the partial sum of its N_b elements and store this interim result to a separate location in global memory that has to be further processed afterwards.

[8] possibly requires padding with zeros

Algorithm 4.9: Matrix Transpose (`mTrans`)

```
Input:
  X:  input matrix
  Δb: block size
```

```
1   // initialization
      shared complex shX[Δb][Δb+1]
3
    // load data into shared memory
5     shX[ty,tx] = X{by,bx}Δb[ty,tx]
      synchronize
7
    // write to XT
9   Z{bx,by}Δb[ty,tx] = shX[tx,ty]
```

```
Output:
  Z = X^T
```

Algorithm 4.10: Matrix Normalization (`mNorm`)

```
Input:
  X: input matrix (interpret as one-dimensional array)
  p: array containing partial sums p_i
  η: normalization parameter
```

```
    // initialization
2     shared float ψ

4   // compute ψ
      if tx==0 then
6       ψ = (p[by])^(η/2)
      end
8   synchronize

10  // normalize X
      int idx = bdx * gdx * by + bdx * bx + tx
12    X[idx] = X[idx] * ψ
```

```
Output:
  X: normalized matrix
```

The index calculation in line 11 of Algorithm 4.10 for the normalization of `X` in line 12 is slightly weird: For `X` $= A^T$, the matrix consists of tiles of size $\Delta s \times N_b N_c$. Blocks of equal block index `by` normalize a single tile. Each block processes Δs consecutive elements of a tile (one per thread) and not a single column, which would be much more intuitive since the number of threads per block equals the number of rows in a tile. Therefore, the memory access is coalesced but the tiled structure is broken, which makes the access appear rather oddly.

For the sake of completeness, we should mention that the regularization of $\hat{V} = A \cdot A^H$ in line 4 of Listing 4.1 is done similarly:

1. Align diagonal elements of $\hat{V}$ linearly in global memory with respect to padding
2. Compute sum of squared norms (reduction)
3. Calculate λ (see line 29 in Algorithm 4.2)
4. Add λ to diagonal elements of $\hat{V}$

4.4 Entire Autocalibration Stage

Algorithm 4.11 outlines how the different kernel programs are finally combined to the whole GRAPPA autocalibration pipeline. To point out that most operations are performed on the device now they are marked with `(d)`.

Algorithm 4.11: GRAPPA Autocalibration Pipeline Using GPGPU (`findWsGPU`)

```
Input:
  S:  ACS data
  GRAPPA parameters (see Symbols in Appendix 6.3)
```

```
1        allocate device memory for matrices and intermediate results w.r.t padding requirements

3        upload S to device

5        for each segment do
   (d)     set up matrices A and B using S
7
           // normalize A and B
9  (d)       transpose A and B
   (d)       realign Bᵀ to power of 2 per segment block
11 (d)       reduction on 2ᵏ aligned Bᵀ: sum of squared norms
   (d)       normalize Aᵀ and Bᵀ
13 (d)       transpose Aᵀ and Bᵀ

15 (d)     U = B Aᴴ

17         // V = (A Aᴴ - λ E)⁻¹
   (d)       V̂ = A Aᴴ
19 (d)       realign diagonal elements of V̂ linearly to power of 2
   (d)       reduction on Diag(V̂): sum of squared norms
21 (d)       regularize V̂

23           download V̂_reg (regularized V̂) to host memory
             V = V̂_reg⁻¹
25           upload V to device memory
```

```
27 (d)    W = U · V

29        download W to host memory
```

```
Output:
  W: reconstruction coefficients
```

There is a CUDA and a CUBLAS version of `findWsGPU`. The former uses `mmul4` for matrix multiplication and `mmul5` for the special case in line 18, the latter is based on `cublasCgemm()` but can not benefit from the Hermitian symmetry of $\hat{V}$ as already mentioned in Section 4.2.4. If B and A are stored back-to-back in global memory, however, a single `cublasCgemm()` call can process both multiplications in lines 15 and 18:

$$\begin{array}{ll} 15: & U = B \cdot A^H \\ 18: & \hat{V} = A \cdot A^H \end{array} \quad \Longleftrightarrow \quad \begin{bmatrix} U \\ \hat{V} \end{bmatrix} = \begin{bmatrix} B \\ A \end{bmatrix} \cdot A^H \; .$$

Since the advantages of graphics hardware become apparent particularly for large input data, this methodology improves the overall performance.

Needless to say, that the reconstruction weights do not necessarily have to be transferred back to host memory (line 29) if no further host-based post-processing is necessary and the subsequent reconstruction stage is executed on the device, too [20].

Chapter 5

Results

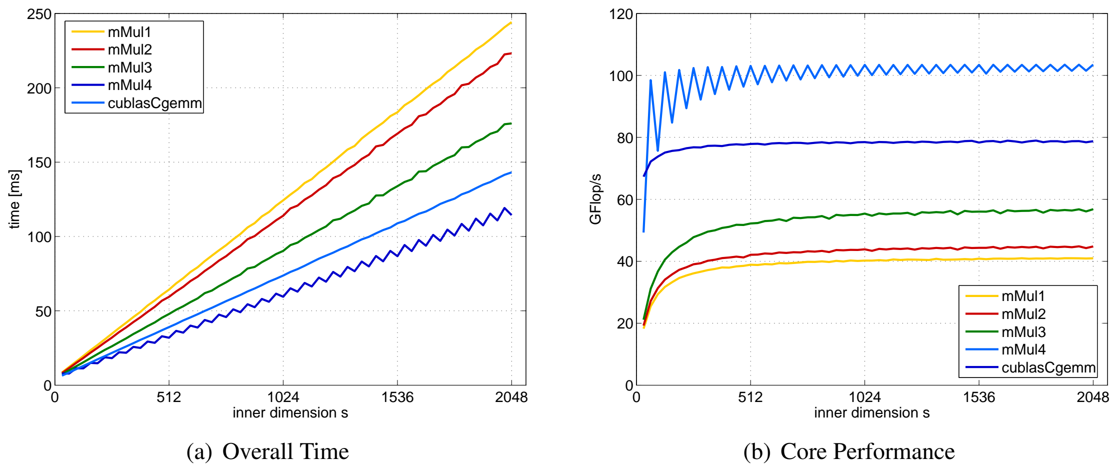
Finally, we compare our GPU-accelerated approaches to current CPU-based implementations based on the following hardware configurations for benchmark tests and validation:

1. Intel Core 2 6700 dual-core CPU at 2.66 GHz, 2x2 MB of L2 cache, 2 GB of RAM (333 MHz), NVIDIA GeForce 8800 GTX GPU with CUDA 1.1 driver, OpenSuse 10.3 (Linux)
2. 2x Intel Xeon 5150 dual-core CPU at 2.66 GHz, 2x2 MB of L2 cache, 4 GB of RAM (333 MHz), NVIDIA GeForce 8800 GTX GPU with CUDA 1.1 driver, Windows XP Professional 64-bit Edition

Diagrams and figures refer to these machines as CPU1/GPU1 and CPU2/GPU2 respectively. On the Windows machine we used *Microsoft Visual Studio 8 (MSVC8)* to compile 32-bit binaries with enabled compiler flags `/Ox /fp:fast /SSE2`. Accordingly, the GNU C Compiler `gcc` (version 4.1.2) was used on the Linux machine merely with `-O3 -funroll-loops -mfpmath=sse2 -msse2 -ffast-math -mtune=prescott`. We want to point out that the benchmark tests are carried out with single-threaded CPU executables unless explicitly mentioned otherwise. Moreover, it is important to know that the graphics hardware has to be initialized before the very first CUDA call, which takes approximately 500 ms. As this has to be done only once for all subsequent computations and can easily be hidden in practical applications, the initialization time is excluded from time measurements in contrast to allocation and transfer times even if they are not directly concerned with the core computation but indispensable. These secondary operations are solely ignored for core performance considerations.

5.1 Matrix Multiplication

First of all, we want to compare the different GPU implementations for complex-valued matrix multiplication of Section 4.2 to get an impression of how the performance is influenced by different optimizations. The parameter configuration of the CUDA kernels `mmul[1-4]` is $\Delta b = 16$, $\Delta b_x = \mu = 4$, and $\Delta b_y = \Delta b_k = 64$ for the multiplication of complex-valued matrices with $r = t = 1024$ and variable inner dimension s according to Equation 4.6. It is remarkable that the basic approach already yields



(a) Overall Time (b) Core Performance

Figure 5.1: Comparison of `mmul[1-4]` and `cublasCgemm()` for fixed $r = t = 1024$ on GPU1.

remarkable performace gains. In spite of increased resource requirements (shared memory and registers) and reduced occupancy, `mmul2` performs 8% better due to enhanced data sharing. Avoiding bank conflicts in `mmul3` saves another 20% in this case. The jagged shape of `mmul4` is caused by additional operations to cancel padding bytes. They are required as the measurements are taken with a step size of 32 but the inner dimension has to be a multiple of $\Delta b_y = 64$. It performs more than 50% better than the basic approach and even 20% better than CUBLAS for this configuration. We will see, however, that things look completely different particularly for smaller input matrices as `mmul4` is better suited for “large” matrices.

The proper computation of the core performance in Figure 5.1(b) requires ptx code analysis to get the number of FP operations contributing to the actual matrix product. The NVIDIA compiler maps a single iteration of the inner product to five low-level instructions (three times `madd`, one `mul`, and one `sub`). The overall costs of the matrix multiplication are therefore given in Equation 4.5 with $\vartheta = 5$.

The optimized kernel `mmul4` is still memory-bound and therefore reaches only 30% of the theoretical peak performance of GPU1.

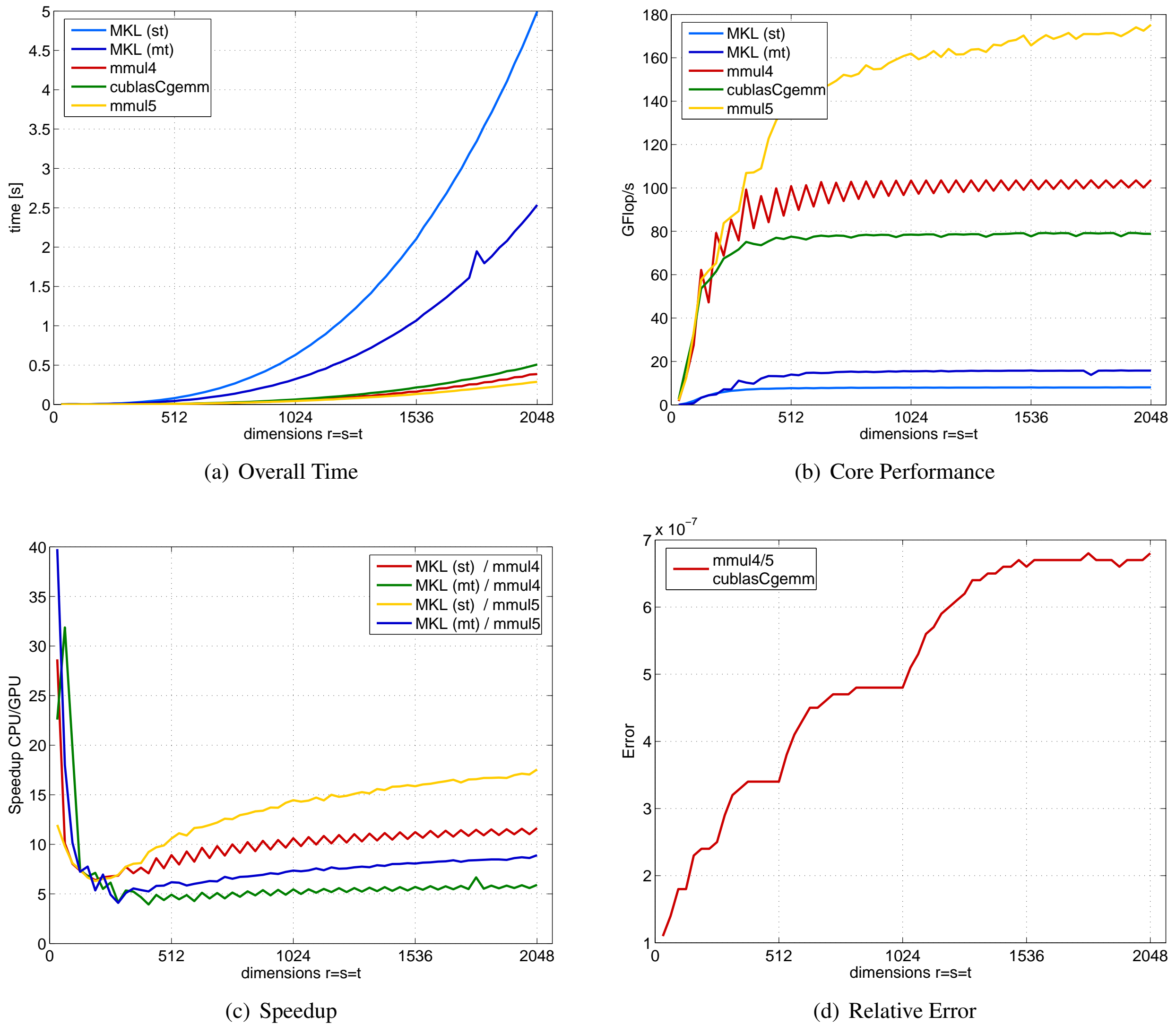
The next experiment opposes the optimized kernels to elaborate CPU-based implementations for matrix multiplication. We used the BLAS Level 3 operation `cblas_cgemm()` provided by the *Intel Math Kernel Library 10.0 (MKL)* [37] with built-in parallelism to obtain excellent scaling on multi-processors. The multi-core ready math routines are threaded using *OpenMP* [66] and highly optimized for Intel processors. The C code was compiled with the *Intel C++ Compiler Professional Edition for Linux* [36] with additional optimization flags for CPU1 (`-march=core2 -axT -xT`) yielding 5 to 10% performance gain compared to the GNU compiler. Figure 5.2 lists the results of the benchmark test for square input matrices of different size.

The overall time increases cubically with the dimension as expected according to Equation 4.5 but at different rates. Obviously, MKL scales quite well for the dual-core architecture of CPU1 if we compare the curves for multi-threaded and single-threaded execution in Figure 5.2(a). The core performance of `cublasCgemm()` and `mmul4` is again limited by approximately 80 GFlop/s respectively 100 GFlop/s as before. Since the Hermitian structure of $A \cdot A^H$ requires only half the number of operations, the core performance of `mmul5` is more than doubled compared to the general CUBLAS approach, 60 to 70% better than `mmul4`, and still increases for larger matrices. Unfortunately, neither `cublasCgemm()` nor `cblas_cgemm()` are able to handle this special case. All in all, the kernels perform 11 (`mmul4`) through 17 times (`mmul5`) faster than the single-threaded MKL approach. For accuracy verification the relative error measure according to Equation 5.1 is used to compare output matrices Z and $\hat{Z}$:

$$\Delta(Z,\hat{Z}) = \frac{1}{rt}\sum_{i=1}^{r}\sum_{j=1}^{t}\frac{\|Z(i,j)-\hat{Z}(i,j)\|}{\|\hat{Z}(i,j)\|} \quad . \tag{5.1}$$

Considering the fact that single-precision FP operations have about six decimal digits of precision[1], the relative error in Figure 5.2(d) is negligible.

[1] actually $log_{10}(2^{23}) \sim 6.92$ digits of precision



(a) Overall Time

(b) Core Performance

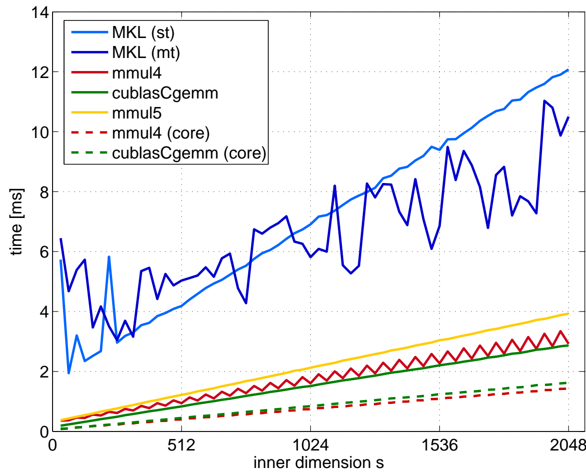
(c) Speedup

(d) Relative Error

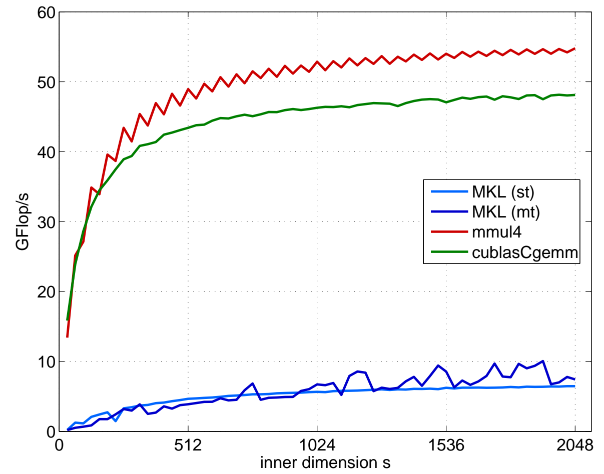
Figure 5.2: Comparison of `mmul[4,5]` and `cublasCgemm()` to single-threaded (st) and multi-threaded (mt) Intel MKL `cblas_cgemm()` for $r = s = t$ on CPU1/GPU1.

Particularly with regard to the algorithms' field of application, the next comparative test chooses matrix dimensions according to usual parameters for GRAPPA autocalibration as listed in Table 4.1. Current MR systems usually work with up to 32 input channels and small acceleration factors. The resulting matrices A and B have only few rows but many columns. Let us assume $N_c = 32$, $N_b = 4$, $R = 5$ and hence $r = t = 128$. The results for this configuration are presented in Figure 5.3.

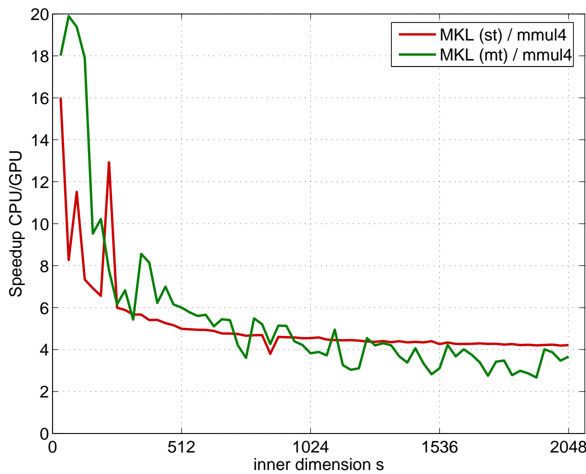
Our kernel programs are not able to harness the massively parallel performance potential of graphics hardware in this situation as only 64 blocks can be used for the execution of `mmul4`, which is much



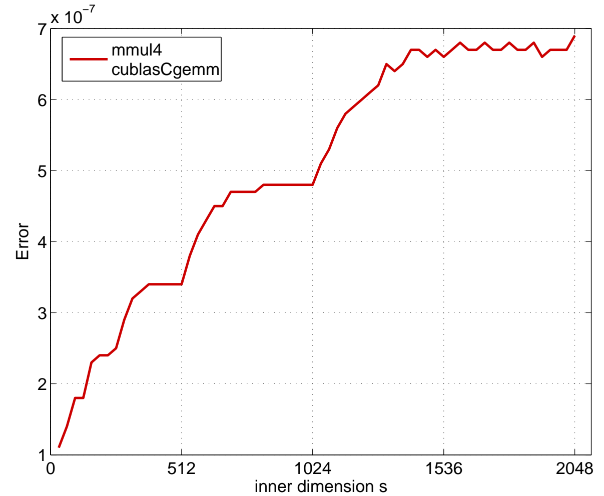
(a) Overall Time



(b) Core Performance



(c) Speedup



(d) Relative Error

Figure 5.3: Comparison of `mmul[4,5]` and `cublasCgemm()` to single-threaded (st) and multi-threaded (mt) Intel MKL `cblas_cgemm` for $r = t = 128$ on CPU1/GPU1.

too less to hide latency. This configuration is even worse for `mmul5` executed by 36 blocks. The core performance as well as the tremendous speedup are reduced by 40 to 60% compared to Figures 5.2(b) and 5.2(c).

We tried to split up the computation of the inner product into several parts to get more blocks. All intermediate matrices though have to be written back to global memory before they can be merged (add entries element-by-element) as there is no possibility for synchronized write operations to global memory for threads of different blocks in CUDA. The resulting performance was even worse.

Another problem of small input data is the overhead for memory allocation and data transfers between host and device. In this case, the ratio of core computation time (dashed line) and overall time (solid line) in Figure 5.3(a) is 50 to 60% in contrast to 10 to 20% for larger input data in Figure 5.2. In terms of core time, the optimized kernel `mmul4` is faster than `cublasCgemm()` in contrast to the overall time as additional time is spent on canceling out padding bytes for `mmul4`, which is excluded from core time. We explicitly mention this fact since padding bytes have to be set to zero in the final GRAPPA autocalibration implementation, anyway, e.g. for matrix transpose operations, no matter which approach is used for matrix multiplication. This requires an initial `cudaMemset()` call whose runtime virtually does not change due to few additional padding bytes potentially required for `mmul4`. As the measurements show, the multi-threaded MKL approach does not scale as well as for large input data. The oscillating curve progression is not an error in measurement but reproducible. It is remarkable that the core performance of both the single- and multi-threaded approach hardly differ at all.

5.2 Matrix Inversion

Besides matrix multiplication, the matrix inversion in step 4 of the GRAPPA autocalibration in Listing 4.1 on page 47 is the second time-critical operation involved as we will see in the next section. The time for CPU-based inversion of $\hat{V} = AA^H - \lambda E$ far outweighs the total runtime of the GPU-accelerated autocalibration stage (see Figure 5.5) even if the matrix to invert is rather small, for instance 128×128 for $N_b = 4$ and $N_c = 32$. Therefore, we tried to translate the inversion algorithm to the graphics hardware.

The simplest approach to invert matrices is the *Gaussian elimination* with partial pivoting to improve the numerical stability of the algorithm. However, this is not the best approach in this case since $\hat{V}$ is a Hermitian matrix. For this reason, it is possible to apply the *Cholesky Decomposition* [17] described in algorithm 5.1 to solve the linear equation system

$$\hat{V}^T \; W^T \quad = \quad B^T$$

for W^T by forward and backward substitution.

Algorithm 5.1: Matlab Implementation for Outer Product Cholesky Decomposition (`cholDec`)

```
Input:
  X: Hermitian matrix
```

```
n=size(X,1);

% init L with lower left part of A
L=zeros(n);
for i=1:n
    L(i:n,i)=X(i:n,i);
end

for k=1:n
    % normalize k^th column
    L(k:n,k)=L(k:n,k)/sqrt(L(k,k));

    % outer product iteration
    for i=(k+1):n
      L(i:n,i)=L(i:n,i) - L(i,k)*L(i:n,k);
    end
end
```

```
Output:
  L: lower triangular matrix with X=L*L^H
```

It is difficult to parallelize the Cholesky Decomposition. The normalization in line 11 has to be done before the outer products are computed in line 15 to adjust the remaining unprocessed elements. In the same way, it is not possible to execute the outer for-loop over k (line 9) in parallel. Therefore, the corresponding CUDA kernel is executed by a single block of n threads. In this way, the kernel does not use the MPs of the graphics hardware to full capacity. Moreover, there is no possibility to benefit from shared memory or texture cache effects for the outer product iteration. The same problems concerning insufficient synchronization functionality also arise for the Gaussian elimination.

The second attempt uses a *Simultaneous Algebraic Reconstruction Technique (SART)* for the computation of the inverse. This technique is originally used for CT image reconstruction [41]. Basically, SART iteratively solves a system of linear equations

$$A\,\vec{x} = \vec{b} \tag{5.2}$$

with known $A \in \mathbb{C}^{n\times n}$, $\vec{b} \in \mathbb{C}^n$, and unknown $\vec{x} \in \mathbb{C}^n$. The main idea is to interpret the elements $a_{i,k}$ of each row i in A as the coefficients of the hyperplane H_i:

$$H_i := \left\{ \vec{t} \in \mathbb{C}^n \,\middle|\, \sum_{k=0}^{n-1} a_{i,k} \cdot t_k = b_i \right\} \ .$$

Assuming that there is a unique solution vector $\vec{x}'$ for Equation 5.2 it corresponds to the intersection point of all hyperplanes:

$$\vec{x}' = \bigcap_{0 \le i < n} H_i \ .$$

The basic version of SART follows the iterative scheme in Equation 5.3 to get an estimation for $\vec{x}'$:

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \frac{1}{n} \sum_{i=0}^{n-1} \frac{b_i - A(i,:)\vec{x}^{(k)}}{A(i,:)A(i,:)^T} A(i,:)^T \ , \tag{5.3}$$

where k denotes the iteration index and $A(i,:)$ is the ith row vector of A. Geometrically considered, this scheme projects the current estimation $\vec{x}^{(k)}$ on each hyperplane and uses the mean of all projected points as next estimation $\vec{x}^{(k+1)}$. The projections to the hyperplanes can be computed simultaneously as they are independent of each other. The initial guess can be chosen for example as $\vec{x}^{(0)} = [1,1,\ldots,1]^T$ if there is no further prior knowledge. A detailed discussion of convergence properties is given in [40].
On the other hand, inverting a matrix A is nothing else but solving

$$A \ \vec{x}_i \ = \ \vec{e}_i \quad \forall 0 \le i < n \, ,$$

where $\vec{e}_i$ is the ith canonical unit vector and $\vec{x}_i$ the ith column vector of A^{-1}. The implementation maps the computation of a single $\vec{x}_i$ to one block of n threads computing a single element of $\vec{x}_i$. Unfortunately, the iterative scheme involves many inner products of length n that have to be computed per thread, which is very slow even for small matrices due to extensive global memory read operations.
It is easily possible, indeed, to use a binary-tree approach to parallelize these computations similarly to the reduction operation described in Section 4.3.2. In this case, each block computes a single element of $\vec{x}_i$. This layout, however, requires additional synchronization between blocks since the kth iteration for $\vec{x}_i$ has to be done before the $k+1^{\text{st}}$ one starts, which is not supported in CUDA. The synchronization between blocks can be avoided, however, by writing back intermediate results per

block and sequentially calling the same kernel for each iteration. First of all, we have implemented a prototype in MATLAB (Mathworks Inc., USA). The SART implementation needs approximately 9500 iterations for matrices of size 128×128 per column to obtain an estimation with four digits of precision. Accordingly, the SART-kernel has to be called 9500 times, too. The host-based inversion of the 128×128 matrix based on Gaussian elimination with partial pivoting takes approximately 18 ms. On the other hand, calling an "empty" kernel 9500 times without any arguments and without any shared memory requirements already takes more than 60 ms. In other words, this overhead rules out the parallel computation of the inner products.

All in all, our CUDA implementations for matrix inversion on GPU are much slower compared to CPU-based algorithms due to the lack of "intra-block" synchronization in CUDA.

5.3 Entire Autocalibration Stage

Finally, we compare the GPU-based autocalibration to the *C++* GRAPPA implementation for image reconstruction in current Siemens MR systems.

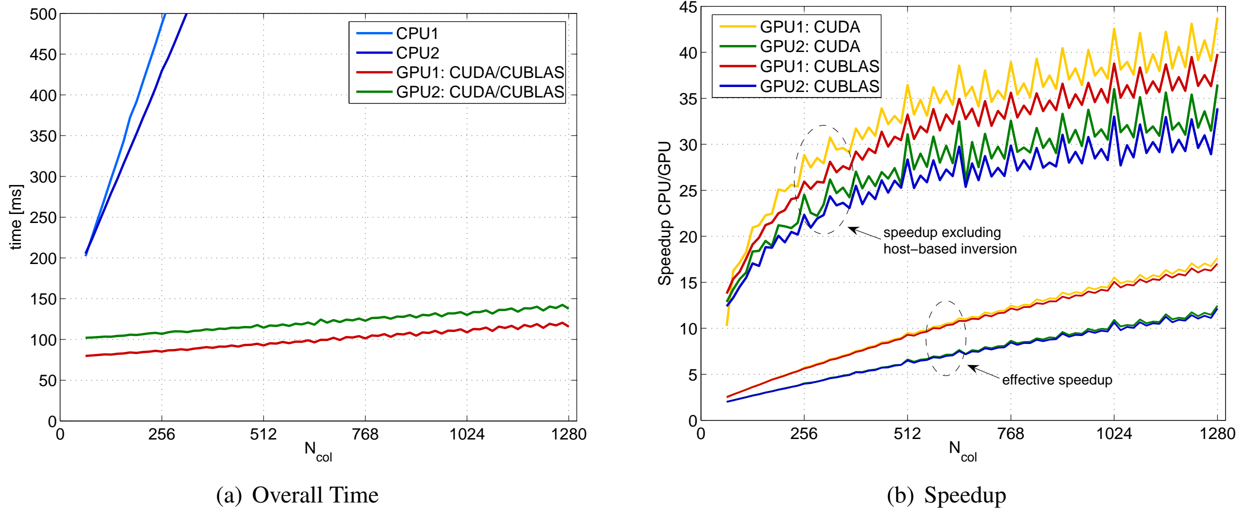
First of all, some general remarks concerning the subsequent benchmark tests. There are many parameters involved in the GRAPPA reconstruction and autocalibration pipeline with different effects on the computational complexity and problem size. In the following, the set of default parameters listed in Table 5.1 is taken by default if nothing else is explicitly mentioned. In order to guard against misunderstandings we want to mention that the kernel programs have not been fine-tuned for this special parameter set but work for virtually arbitrary parameter values.

5.3.1 Standalone Autocalibration

For a start, we extracted a standalone version of the relevant autocalibration routines to work independently of the whole reconstruction pipeline with random input data. As the reconstruction time is almost constant for each slice image of an MR scan, all timings are based on the computations for a single slice image. Moreover, we would like to emphasize at this point that the performance of algo-

Description	Symbol	Default Value
Image Columns	N_{col}	512
Input/Output Channels	N_c	32
Acceleration Factor	R	4
Reference Lines	N_r^{ACS}	24
Block Size	N_b	4
Segment Length	Δs	128
Regularization Parameter	χ	0.0001
Normalization Parameter	η	1.0

Table 5.1: Default Values of GRAPPA Parameters



(a) Overall Time

(b) Speedup

Figure 5.4: Results for `findWs[GPU]` Depending on N_{col}

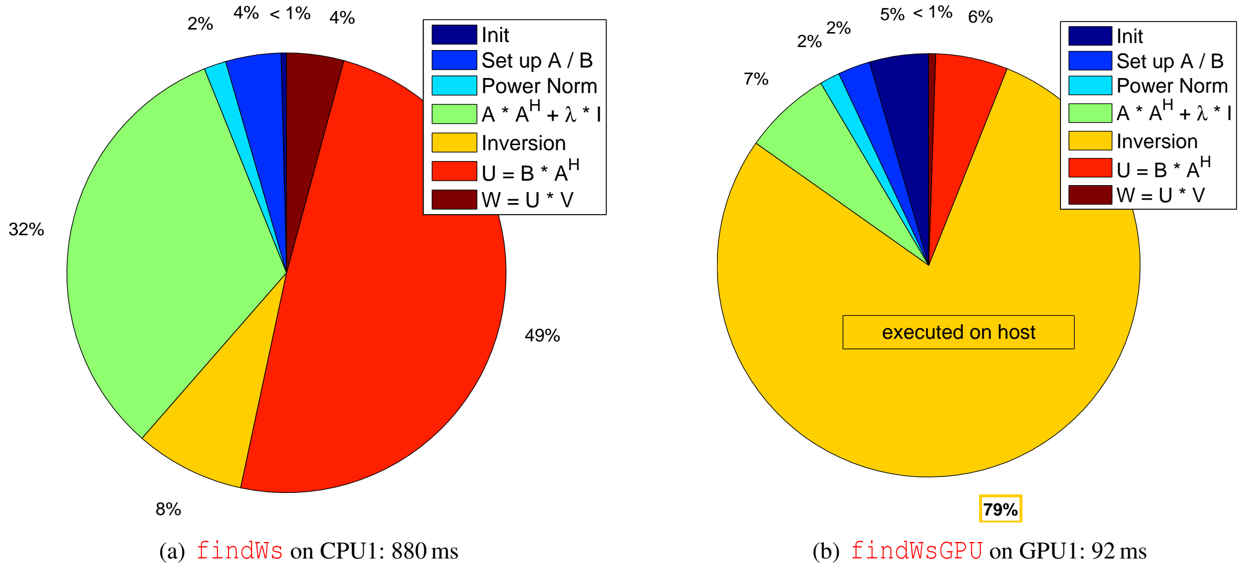
rithms for matrix computation implemented in the CPU-based autocalibration stage is far from being comparable with highly optimized libraries such as Intel MKL.

In a first test run all parameters are fixed except the image width. Figure 5.4 illustrates the results for variable number of image columns.

The overall time for the autocalibration linearly increases with N_{col} as already deduced in Table 4.2. The curves of the CPU-based approaches can easily be extrapolated as the slope remains constant (512: 758 ms / 881 ms, 1024: 1418 ms / 1688 ms, 1280: 1716 ms / 2044 ms). Although both machines feature equal graphics cards, the performance apparently differs most probably because of differences

in the hardware drivers and different operating systems. We can virtually rule out a connection with different CPU or C compiler configurations since CPU1 performs worse compared to CPU2.

The total time of the CUDA and CUBLAS version is almost the same for this configuration, which results in equal effective speedups up to 17.5 on GPU1. Analyzing the timings for the different steps in more detail reveals the bottleneck of the GPU-powered approaches as shown in Figure 5.5.



(a) `findWs` on CPU1: 880 ms (b) `findWsGPU` on GPU1: 92 ms

Figure 5.5: Breakdown of Overall `findWs[GPU]` Execution Time for $N_{\text{col}} = 512$

Obviously, matrix multiplication is the most time-consuming and compute-intensive task in `findWs` (85%). By contrast the bottleneck of the GPU-based execution is definitely the matrix inversion (79%) that is still executed on the host. The absolute time for the inversion is of course the same for both. If the time for matrix inversion is excluded from speedup computations for the moment, the speedup factor reaches 43 (36) on GPU1 (GPU2) according to Diagram 5.7(b). The zigzag shape of the curves results from extensive padding required for the different kernels. Section 4.3 described the initialization of A and B per k-space segment. If the segment size is a multiple of 16 (half-warp size) we achieve best performance in terms of memory throughput due to coalesced memory operations in each half-warp. As there are $\Delta s = 4$ segments of size $N_{\text{col}}/4$, we expect best performance for N_{col} being a multiple of 64, which is obviously validated in Diagram 5.6.

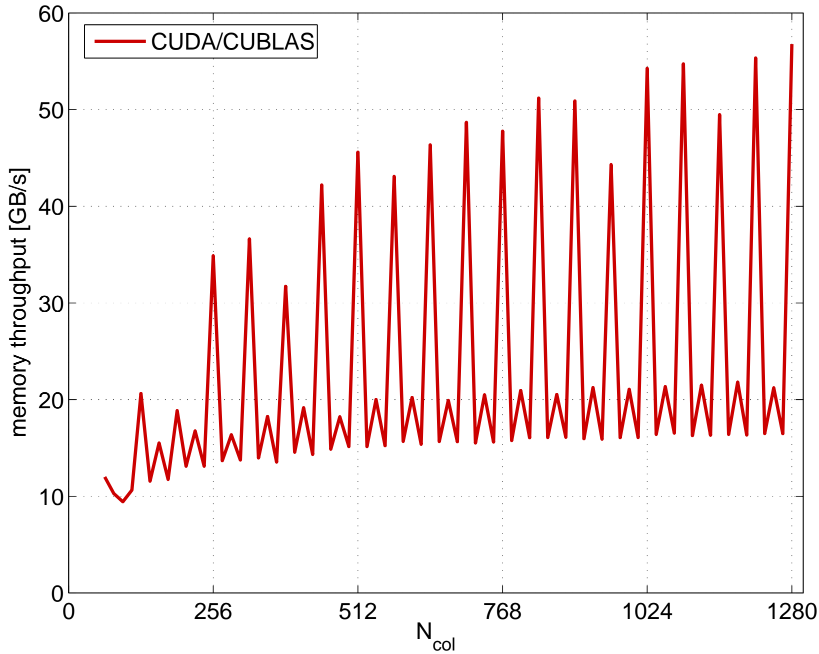
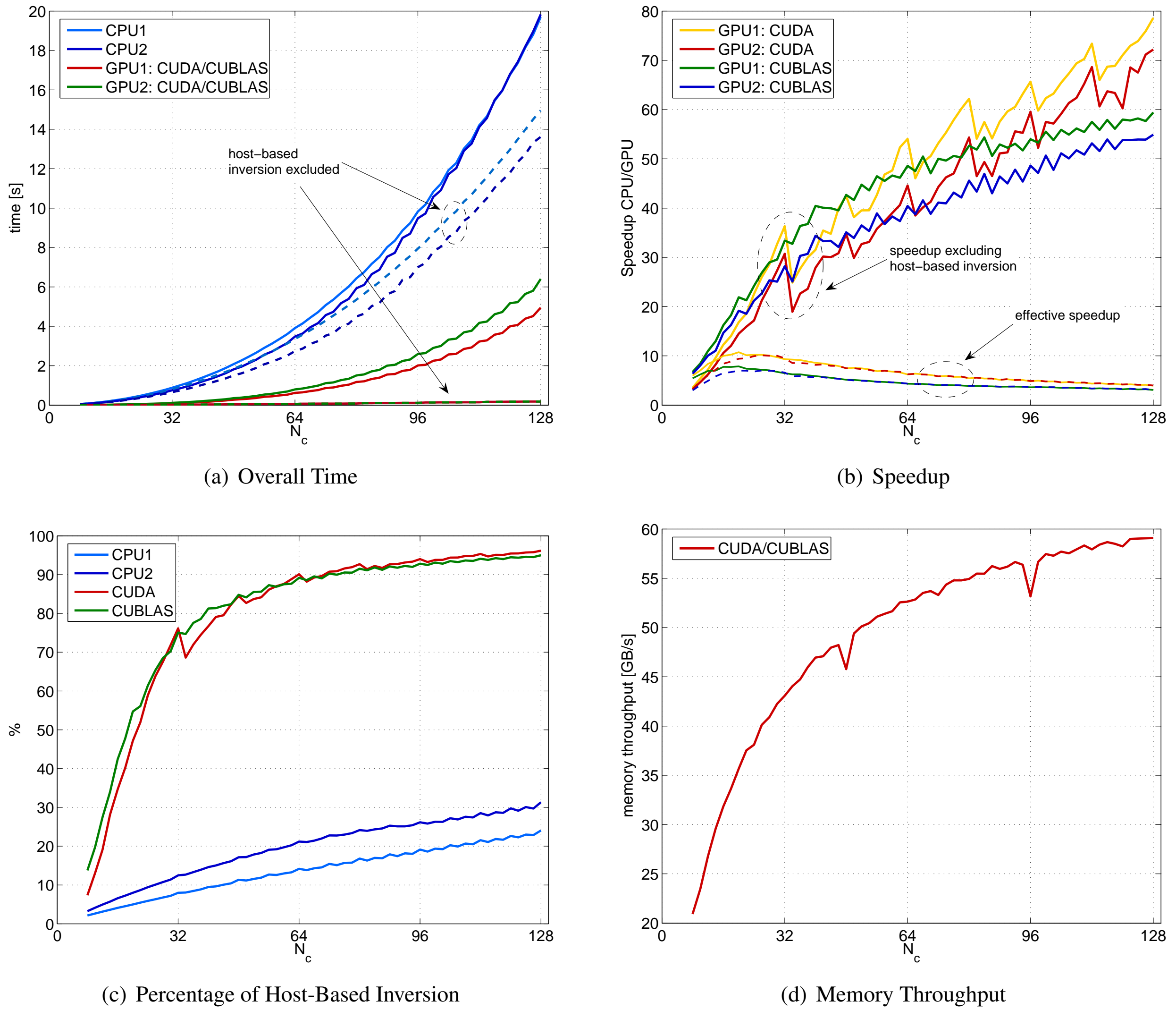


Figure 5.6: Effective Memory Throughput During Initialization of A and B

For multiples of 64 and $N_{\text{col}} \geq 512$, the initialization of A and B reaches an effective memory throughput in the range of 45 to 56 GB/s and hence approximately 60% of the theoretical memory bandwidth.

As already discussed in Section 4.1.3, the number of channels is a limiting factor since it is cubically involved in the overall computational costs of the autocalibration stage (see Table 4.2). For this reason, the performance of our GPU-based approach for more than 32 channels is a matter of particular interest with regard to future MR scanners that will work with arrays of up to 128 coils currently used in some exploratory systems. The results for varying number of channels and fixed $N_{\text{col}} = 512$ are illustrated in Figure 5.7.

The total time of the CPU-based autocallibration reaches almost 20 seconds for the computation of the reconstruction coefficients for a single (!) slice, which is unacceptable for practical applications especially as the required time for the remaining reconstruction stages increases in the same manner! The "inversion problem" is even worse in this case as shown in Diagram 5.7(c). More than 90% of the overall time of the GPU-accelerated approach is spent on matrix inversion for more than 64 channels. This is why the effective speedup in Diagram 5.7(b) continuously decreases to four for large N_c.



(a) Overall Time

(b) Speedup

(c) Percentage of Host-Based Inversion

(d) Memory Throughput

Figure 5.7: Results for `findWs[GPU]` Depending on N_c

Excluding the inversion and comparing only parts of the autocalibration that have actually been mapped to GPU, yields speedups between 50 and 80 for more than 64 channels on GPU1. Moreover, the diagram points out that the CUDA version (`mmul4`, `mmul5`) performs 15 to 25% better than the CUBLAS implementation for $N_c \geq 64$.

Finally, Diagram 5.7(d) shows that the memory bandwidth is used much better by the initialization kernel as the number of executed thread blocks similarly increases with the number of channels, which causes much better occupancy and better latency hiding.

The relative error adds up to 10^{-4} for the channel and column test run as there are involved sev-

eral consecutive matrix multiplications and the numerically ill-conditioned matrix inversion causing considerable error accumulation. There are even remarkable discrepancies between the results of the host-based implementation for matrix inversion depending on the compiler.

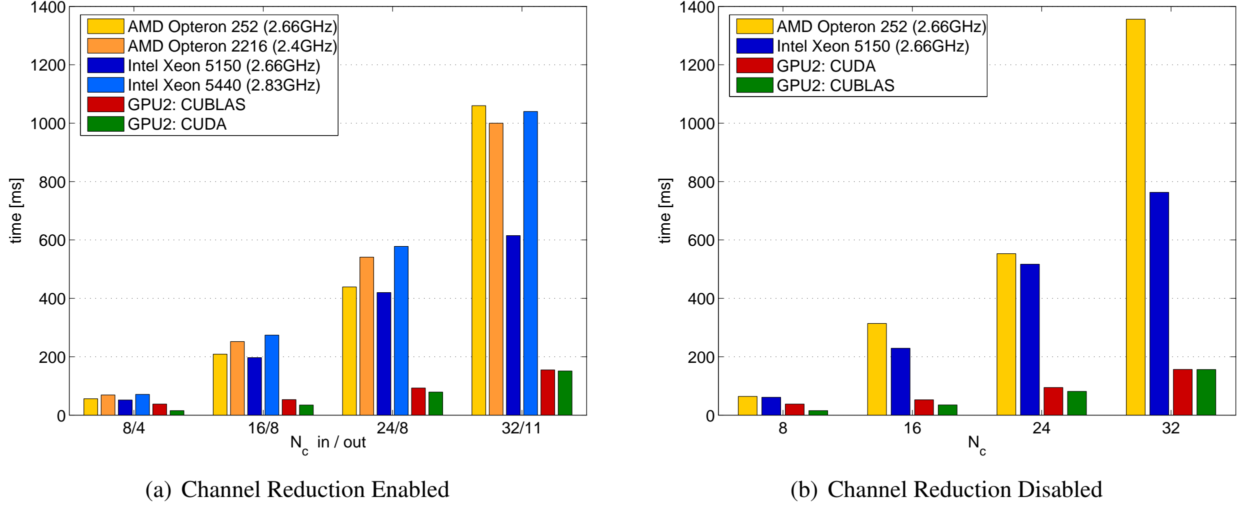
In the beginning we met problems concerning limitations of device memory as in the proceeding of our algorithm additional memory space is needed to separately store transposed and reordered matrices, which allows for efficient access patterns in some kernels. The required memory space, therefore, exceeds the actual size of ACS data, A, and B by far.
We have overcome these difficulties by implementing a smart memory management avoiding redundancy whenever possible. As a result, this technique extends the range of possible parameter values the graphics hardware can cope with. A few configurations and the corresponding GPU memory requirements of `findWsGPU` are listed in Table 5.2.

#	N_{col}	N_c	R	N_r^{ACS}	N_b	Δs	Memory
1	512	32	4	24	4	128	10.2 MB
2	1024	128	4	24	4	256	81.8 MB
3	1024	64	16	128	8	256	197 MB
4	1024	64	16	256	16	256	298 MB
5	1024	128	8	48	8	256	717 MB

Table 5.2: GPU Memory Requirements of `findWsGPU` for Different Configurations

5.3.2 Integrated Autocalibration

In a final step, we integrated the GPU-powered implementations for the autocalibration stage into the entire reconstruction pipeline. Siemens Medical Solutions kindly provided practical test data and corresponding timings for different image reconstruction host systems. The reconstruction pipeline is part of a complex software system that is currently compiled with *Microsoft Visual Studio 6 (MSVC6)*. The software and the final application binaries support multi-threaded execution on multi-core processors. The following time measurements refer to either single-threaded execution or the time per (single) core during multi-threaded execution. The autocalibration parameters are chosen as listed in Table 5.1 except for $N_{\text{col}} = 1024$, $R = 2$, and $\Delta s = 204$.



(a) Channel Reduction Enabled (b) Channel Reduction Disabled

Figure 5.8: Comparison of GPU to CPU for Practical Test Cases

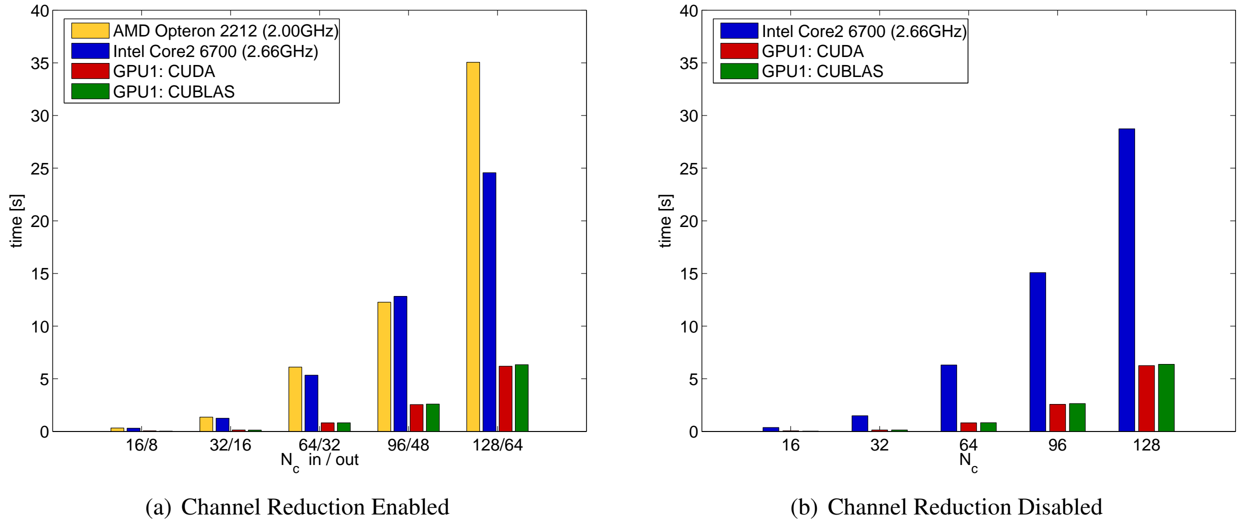
The Intel Xeon 5440 system is equipped with low frequency DRAM, which is the reason for the worse performance compared to the Intel Xeon 5150 of lower core clock.

Both GPU implementations perform four to nine times faster than the reference CPU implementation. The Intel Xeon 5150 system corresponds to CPU2 (host system of GPU2) and is the fastest listed system for all configurations (GPU speedup of four to five). This is worse compared to the situation in Figure 5.4 for $N_{\text{col}} = 1024$ due to the different configuration. First of all, the segment size is no longer a power of 2 ($\Delta s = 204$), which reduces the memory throughput during matrix initialization to approximately 10 GB/s. Even worse are the matrix dimensions of A and B in this situation according to Table 4.1[2]: $A \in \mathbb{C}^{128\times 3712}$, $B \in \mathbb{C}^{32\times 3712}$ for $N_c = 32$ in contrast to the configuration used in Figure 5.4 ($N_{\text{col}} = 1024$): $A \in \mathbb{C}^{128\times 3072}$, $B \in \mathbb{C}^{96\times 3072}$. The time for the inversion of $A \cdot A^H \in \mathbb{C}^{128\times 128}$ remains the same. The computation of the inner products for the matrix multiplications $U = B \cdot A^H$ and $\hat{V} = A \cdot A^H$, however, requires more arithmetic operations. On the other hand, B has fewer rows, which reduces the number of blocks that are executed for `mmul4` to compute U and $W = U \cdot V$ from 48 down to 16 (64 threads each). In other words, each of the 16 multiprocessor executes exactly one block but there is no possibility to schedule different blocks for better latency hiding.

The same holds true for the comparison of enabled and disabled channel reduction which affects only the number of rows in B. In the case $N_c^{\text{in}} = 32$ and $N_c^{\text{out}} = 11$ in Diagram 5.8(a), for example, the

[2]additional padding requirements are missing in Table 4.1

number of rows is cut by half from 32 to 16. Therefore, only eight blocks are used for the computation of U and W, which is much too less to keep all 16 multiprocessors busy. This is the reason why the overall time GPU2 virtually does not change for disabled channel reduction in Diagram 5.8(b) in contrast to the CPU timings. The GPU-based implementation can benefit from channel reduction only for larger values such as $N_c^{\text{in}} = 128$, $N_c^{\text{out}} = 64$. Nevertheless, Diagram 5.7(c) shows that the computation of the inverse of $\hat{V}$ is the actual bottleneck particularly for large matrices. The time for inversion itself is unaffected as a smaller number of output channels does not reduce the dimensions of $\hat{V}$. Therefore, there is virtually no noticeable difference between GPU timings in Diagrams 5.9(a) and 5.9(b) even for 128 input channels.



(a) Channel Reduction Enabled

(b) Channel Reduction Disabled

Figure 5.9: Outlook on Future Number of Channels

5.3.3 Computational Error

According to Diagram 5.3(d) the error for a single matrix multiplication is of magnitude 10^{-7} to 10^{-6}, which is in accordance to single-precision 32-bit accuracy currently supported by recent graphics hardware. Moreover, we have already mentioned that the relative error of the final reconstruction weights adds up to 10^{-4} because of several consecutive matrix multiplications and the numerically ill-conditioned matrix inversion in particular. The computation of the error metric according to Equation 5.1 is based on reference data obtained from the CPU-based autocalibration algorithm that is

compiled with enabled `/fp:precise` flag for precise arithmetic and higher accuracy. The relative error decreases by approximately one order of magnitude if fast FP arithmetic is used (`/fp:fast`) to compute reference data.

We also compared the results of 191 test cases each containing a set of 12-bit greyscale slice images to examine the impact of the “inaccurate” reconstruction weights on final reconstructed MR images. The reference images are generated by the original MSVC6-compiled reconstruction pipeline of Siemens Medical Solutions. We encountered relative errors up to 10^{-2} that are caused by the disparate behavior of the inversion algorithms compiled with MSVC6 respectively MSVC8 even for the same input. In spite of the relative deviation up to 10^{-2} there are no visible differences in the vast majority of reconstructed images as most of the pixels differ by only one or two bit compared to the 12-bit reference images. In the cases of higher deviations we observed localized pixel differences that have been rated by experts as noncritical and sometimes as even “better” than the reference image.

Finally, we combined our GPU-powered algorithms for autocalibration with the GPU-accelerated implementation for efficient k-space reconstruction presented in [20]. The reconstruction algorithm itself only leads to an error in the range of 10^{-5} to 10^{-6} if reference reconstruction weights are used as input. Therefore, the combination of both algorithms generated virtually the same images as the CPU-based reconstruction did using the GPU-accelerated autocalibration library.

Chapter 6

Conclusion

6.1 Summary

The first part of this thesis provided an overview of MRI and explained how the acquisition time can be reduced by parallel imaging techniques such as GRAPPA. GRAPPA in a nutshell: undersample k-space and reconstruct missing information by fitting acquired data to k-space gaps. The reconstruction of missing data is a computationally intensive task and basically consists of two stages: The computation of the fitting coefficients in an initial autocalibration stage and the actual reconstruction of the missing data. These two processes account for approximately 50% of the overall computation time required for the entire image reconstruction assuming current parameter configurations. For a large number of input channels ($N_c \geq 128$) this portion even amounts more than 80%. Obviously, k-space reconstruction is the bottleneck particularly for future configurations.

The second part described the massively parallel and specialized architecture of graphics hardware and how to effectively harness its computational power to accelerate general-purpose computations. Moreover, we outlined the Compute Unified Device Architecture along with the uniform shader model of most recent NVIDIA graphics cards allowing for programming specialized hardware on an abstract layer.

The final part presented different CUDA kernels for complex-valued matrix multiplication on GPU and explained various optimization techniques that have been applied step-by-step yielding speedups of 12 through 18 for special cases compared to the highly optimized Intel MKL. We used

the optimized kernels to speed up the GRAPPA autocalibration stage as the involved matrix multiplications usually make up more than 80% on a CPU. The parts that have been mapped to graphics hardware[1] perform 30 to 45 times faster for current configurations and up to 80 times faster with regard to future parameter values ($N_c = 128$). The computation of the inverse is still executed on the host, which is the new bottleneck. All in all, the GPU-accelerated implementation is 5 up to 18 times faster than the CPU-based GRAPPA autocalibration and now only makes up two to three percent of the overall reconstruction time.

6.2 Future Work

As we have already mentioned several times, the host-based matrix inversion turns out to be the new bottleneck of the optimized GPU implementation since 75% to 95% of the overall time is spent on computing the inverse of $\hat{V} = AA^H$. As long as it is executed on CPU, every attempt to further improve the kernels for matrix multiplication is practically useless, e.g. optimization for small matrices according to [62]. This is why we should concentrate on translating this task to GPU in a next step for further improvements. Speedup factors for matrix inversion are most likely much lower compared to those of matrix multiplication as this problem is hard to parallelize particularly because of restricted communication and synchronization possibilities in CUDA as described in Section 5.2. Moreover, we could run into serious accuracy problems as the inversion is numerically ill-conditioned. It remains to be seen if 32-bit single-precision is enough. Luckily, GPUs with support for double-precision are announced for 2008.

We have also shown that the GPU implementation of the autocalibration stage can easily be combined with the GPU-accelerated GRAPPA reconstruction [20]. The fitting coefficients are computed per k-space segment, however, but not for each column as required by the k-space reconstruction stage. Therefore, the weights are linearly interpolated on CPU in the current implementation. As the texture stage allows for efficient coefficient interpolation on graphics hardware, the intermediate transfer of the reconstruction weights from device to host and back could be saved.
In terms of memory requirements, it is even better to interpolate the weights on-the-fly, i.e. interpo-

[1] whole autocalibration stage except matrix inversion

late coefficients during k-space reconstruction at the time they are needed. If we assume the default parameters in Table 5.1 for example a single coefficient matrix requires 96 kB of global memory. The autocalibration stage computes one matrix of this kind for each of the four k-space segments, i.e. 384 kB of global memory. The interpolated weights for each column, however, need Δs times the amount, i.e. 48 MB. This discrepancy gets even worse for more channels and larger images (same example but $N_c = 128$ and $N_{\text{col}} = 1024$: 6 MB vs. 1.5 GB).

Data transfer between host and device carries more and more weight particularly for future GRAPPA parameters. Luckily, CUDA provides special routines for allocation of *page-locked host memory* that halves the transfer time. Moreover, pinned memory allows for streamed asynchronous memory transfers to hide transfer times by double buffering since CUDA 1.1 [61]. We could not benefit from page-locked memory in our integrated GPU implementation as the application framework has an own embedded management system for host memory.

Another possibility for further acceleration of the GRAPPA algorithm is the final inverse Fourier transform of the reconstructed k-space especially since NVIDIA provides efficient implementations of the according routines included in the easy-to-use CUFFT library [57].

6.3 Final Remarks on CUDA

CUDA allows for programming highly specialized graphics hardware of remarkable computational power on an abstract layer without expert knowledge of graphics APIs such as OpenGL. Basic knowledge of C-programming and the few CUDA extensions to the ANSI C standard are enough for a start to implement own kernels. As demonstrated in Sections 4.2.1 and 5.1 even basic approaches without elaborate optimizations achieve remarkable results. Furthermore, the higher-level libraries CUBLAS and CUFFT provide optimized and easy to use routines for different tasks and will probably be further improved in future releases.

It is quite simple to integrate CUDA routines into complex software systems to leverage the computational power of graphics cards even if it is hardly possible to achieve the entire theoretical peak performance and memory bandwidth. The gain of performance highly depends on the respective problem particularly with regard to parallelizability, arithmetic intensity, as well as memory requirements.

On the other hand, it became clear that detailed knowledge of the CUDA execution model, the `nvcc` compiler, and the underlying hardware is still required for elaborate optimization techniques. Optimizing CUDA kernels is a science in its own right. NVIDIA provides the occupancy calculator [58] and the CUDA Visual Profiler [64] for this purpose. The capabilities of these tools are quite limited, though. As there are many influencing factors it is quite hard to optimize kernels analytically. This is why experimental optimization and tedious trial-and-error methods often successfully reach the goal more quickly. In some cases, just tiny changes of source code unexpectedly have an impact on runtime.

Another drawback is the limited support for debugging CUDA kernels, which is only possible in device emulation mode up to now. Trouble-shooting may become a quite cumbersome task as not all phenomenons of concurrency showing up during execution on the device itself can be emulated properly.

Anyhow,

CUDA pushes “The Democratization of Parallel Computing” and brings it to the masses [43].

Acknowledgements

I would like to thank Prof. Dr. Günther Greiner and Quirin Meyer for their kind supervision of this thesis at the Chair for Computer Graphics, Department of Computer Science 9, Friedrich-Alexander University of Erlangen-Nuremberg.

Particular thanks also go to Michael Peyerl, Gerald Mattauch, and Dr. Swen Campagna from Siemens Medical Solutions, Erlangen for kindly providing the GRAPPA source code and for their helpful advice and assistance with all occurring problems especially during the integration into the complex application framework.

I would like to particularly thank my friend Robert Grimm for the close collaboration, his fruitful hints, and discussions. It was a pleasure to work with you.

Last but not least, I would like to thank all people that reviewed this thesis, namely my father Klaus, my brother Thomas, and my friend Birgit Stiller – I really enjoyed reading your comments!

Thank you.

Appendices

Appendix A

Notation and Preliminaries

A.1 Matrix Structure

We use an abbreviation for matrices of the following form:

$$M = \Big[m(i,j)\Big]_{x \le j < x+\Delta x}^{y \le i < y+\Delta y} = \Big[m_{i,j}\Big]_{x \le j < x+\Delta x}^{y \le i < y+\Delta y} =$$
$$= \begin{pmatrix} m(y,x) & m(y,x+1) & \cdots & m(y,x+\Delta x-1) \\ m(y+1,x) & m(y+1,x+1) & \cdots & m(y+1,x+\Delta x-1) \\ \vdots & \vdots & \ddots & \vdots \\ m(y+\Delta y-1,x) & m(y+\Delta y-1,x+1) & \cdots & m(y+\Delta y-1,x+\Delta x-1) \end{pmatrix} .$$

The superscript part with counter i after the $\big[\cdot\big]$ operator describes the blocked structure of the Δy matrix rows, and accordingly the subscript part with counter j that of the Δx matrix columns.

A.2 Matrices in Memory

From a mathematical point of view, matrices are two-dimensional data structures (m rows, n columns). As the memory of a computer is addressed linearly, however, we have to unfold them[1]. Basically, there are two ways that are commonly used in practise to linearize dense matrices.

[1] The same holds true for two-dimensional arrays.

Row-Major Order The matrix rows are linearly aligned one after the other. The element in the i-th row and j-th column is stored at the linear index position $i \cdot n + j$. This technique is used in many programming languages such as C, C++, and CUDA. It is assumed to be used by default.

Column-Major Order This time, the order is turned upside down. The element in the i-th row and j-th column is stored at the linear index position $j \cdot m + i$, now. This order is used, for example, in Fortran, Matlab, and CUBLAS.

To get rid of the different storage orders, matrices and the corresponding linearized arrays are treated equally. As shown in table A.1, both are addressed as two dimensional arrays in code snippets with a separate index for row and column no matter which storage order is used.

Operation	Code snippet
Initialization	<dataType> M[#rows][#columns]
Access single element	element = M[rowIdx,colIdx]
Access column	element = M[●,colIdx]
Access row	element = M[rowIdx,●]

Table A.1: Accessing Matrices

A.3 Segmented Matrices

In the majority of cases, matrices are divided into tiles in this thesis. Therefore, we introduce a notation for addressing whole submatrices to avoid tedious index calculations. Given a matrix $M \in \mathbb{C}^{m \times n}$ which is segmented into blocks of size $\Delta y \times \Delta x$ with $m/\Delta y, n/\Delta x \in \mathbb{N}$, then

$$M\{i,j\}^{\Delta y}_{\Delta x} := \Big[m(i \cdot \Delta y + p, j \cdot \Delta x + q) \Big]^{0 \leq p < \Delta y}_{0 \leq q < \Delta x} \quad .$$

In other words, $M\{i,j\}$ is the block of M located at the ith row and j^{th} column within the overlayed grid of blocks. If one of the block sizes does not matter or both are equal we only specify the relevant one: $M\{i,j\}_{\Delta x}$ or $M\{i,j\}^{\Delta y}$.

A.4 Implicit Variables

As described in Section 3.3.3, there is a grid-block-thread hierarchy for the execution of a kernel. Each thread is well-defined by the *grid-index* of the associated block along with the *thread-index* within this block. During execution, all threads can access their index tags as well as the according dimensions of grids and blocks provided by the CUDA environment as implicit variables listed in Table A.2.

	CUDA identifier	Our identifier	Description
index	`blockIdx.x`	`bx`	column of block in grid
	`blockIdx.y`	`by`	row of block in grid
	`threadIdx.x`	`bx`	column of thread in block
	`threadIdx.y`	`by`	row of thread in block
dimension	`gridDim.x`	`gdx`	number of columns in grid
	`gridDim.y`	`gdy`	number of rows in grid
	`blockDim.x`	`bdx`	number of columns in block
	`blockDim.y`	`bdy`	number of rows in block

Table A.2: Index and Dimension Tags for CUDA Threads

List of Acronyms

ACS	Auto-Calibration Signal
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARB	OpenGL Architecture Review Board
BLAS	Basic Linear Algebra Subprograms
Cg	C for Graphics
CPU	Central Processing Unit
CT	Computed Tomography
CTM	Close To The Metal
CUDA	Compute Unified Device Architecture
EPI	Echo-Planar Imaging
FFT	Fast Fourier Transform
FID	Free Induction Decay
FLASH	Fast Low-Angle Shot
FOV	Field of View
FP	Floating-Point (Number)
FSE	Fast Spin Echo Sequence
GE	Gradient Echo
GLSL	OpenGL Shading Language
GPGPU	General-Purpose Computing on GPUs
GPU	Graphics Processing Unit
GRAPPA	Generalized Autocalibrating Partially Parallel Acquisitions

HLSL	High Level Shading Language
HPC	High-Performance Computing
madd	Multiply-Add Instruction
MKL	Intel Math Kernel Library
MP	Multiprocessor
MRI	Magnetic Resonance Imaging
MSVC6	Microsoft Visual Studio 6
MSVC8	Microsoft Visual Studio 8
NMR	Nuclear Magnetic Resonance
nvcc	NVIDIA C Compiler
PAT	Parallel Acquisition Technique
PD	Proton Density
PE	Phase-Encoding (Direction)
PILS	Partially Parallel Imaging with Localized Sensitivities
pMRI	Parallel Magnetic Resonance Imaging
PTX	Parallel Thread Execution (Code)
RF	Radio Frequency
RO	Read-Out (Direction)
SART	Simultaneous Algebraic Reconstruction Technique
SDK	Software Development Kit
SE	Spine Echo
SENSE	Sensitivity Encoding
SFU	Special Function Unit
SIMD	Single Instruction, Multiple Data
SMASH	Simultaneous Acquisition of Spatial Harmonics
SNR	Signal-to-Noise Ratio
SPMD	Single Program, Multiple Data
TSE	Turbo Spin Echo Sequence

List of Symbols

α	RF Pulse Flip Angle
χ	Regularization Parameter for GRAPPA Autocalibration
$\Delta\omega_0$	Frequency Bandwidth of G_S
Δk_x	Step Size in PE-Direction
Δk_y	Step Size in RO-Direction
Δs	Segment Length
Δz_0	Slice Thickness
η	Normalization parameter for GRAPPA Autocalibration
γ	Gyromagnetic Ratio
κ	Parameter for Power Clipping in GRAPPA Autocalibration
λ	Normalization Coefficient
ω	Larmor Frequency
ρ	FOV Image
ρ_{PD}	Proton Density
τ	Time Period Between 90 and 180 Degree Pulse
B	Strength of Static Magnetic Field
C_k	Sensitivity Map of Coil k
E	Identity Matrix
G_F	Frequency-Encoding Gradient
G_P	Phase-Encoding Gradient
G_S	Slice-Selection Gradient
I_k	Limited FOV Image of Coil k

K_{PE}	Number of Filter Kernel Rows for Improved GRAPPA Autocalibration
K_{RO}	Number of Filter Kernel Columns for Improved GRAPPA Autocalibration
k_{x_0}	First Column of ACS Segment
k_{y_0}	Row Position of First ACS Block
M_{XY}	Transverse Magnetization
M_{Z}	Longitudinal Magnetization
N_r^t	Number of Registers Available per Thread
N_b^{mp}	Number of Concurrently Processed Blocks per Multiprocessor
N_r^{mp}	Number of Registers per Multiprocessor on GPU
N_t^b	Number of Threads per Block
N_b	Number of Blocks in Filter mask for GRAPPA Reconstruction
N_c	Number of Coils
N_r^{ACS}	Number of ACS Lines
N_{col}	Number of k-space Columns of Each Coil
N_{sh}	Number of Filter Kernel Shifts in RO- and PE-Direction for Improved GRAPPA Autocalibration
$N_{\text{sh}}^{\text{PE}}$	Number of Filter Kernel Shifts in PE-Direction for Improved GRAPPA Autocalibration
$N_{\text{sh}}^{\text{RO}}$	Number of Filter Kernel Shifts in RO-Direction for Improved GRAPPA Autocalibration
N_b^{ACS}	Number of ACS Blocks
R	Acceleration Factor aka Reduction Factor
S	MR Signal Intensity
T_1	Time Constant of Spin-Lattice Relaxation
T_2	Time Constant of Spin-Spin Relaxation
T_2^*	Effective Time Constant of Spin-Spin Relaxation
T_E	Echo Time
T_R	Pulse Sequence Repetition Time
w	Reconstruction Coefficient

Bibliography

[1] Advanced Micro Devices, Inc. ATI CTM Guide, 2006.
http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf.

[2] J. R. Ballinger. MRI tutor - introduction to MRI, March 1997.
http://www.mritutor.org/mritutor/index.html, last accessed 2008/01/12.

[3] M. Blaimer, F. Breuer, M. Müller, R. M. Heidemann, M. A. Griswold, and P. M. Jakob. SMASH, SENSE, PILS, GRAPPA: how to choose the optimal method. *Top Magn Reson Imaging*, 15(4):223–236, Aug 2004.

[4] M. Blaimer, F. A. Breuer, M. Müller, N. Seiberlich, D. Ebel, R. M. Heidemann, M. A. Griswold, and P. M. Jakob. 2D-GRAPPA-operator for faster 3D parallel MRI. *Magnetic Resonance in Medicine*, 56(6):1359–1364, Dec 2006.

[5] M. Blaimer, F. A. Breuer, N. Seiberlich, M. F. Müller, R. M. Heidemann, V. Jellus, G. Wiggins, L. L. Wald, M. A. Griswold, and P. M. Jakob. Accelerated volumetric MRI with a SENSE/-GRAPPA combination. *J Magn Reson Imaging*, 24(2):444–450, Aug 2006.

[6] F. Bloch, W. W. Hansen, and M. Packard. Nuclear induction. *Phys. Rev.*, 69(3-4):127, Feb 1946.

[7] F. Bloch, W. W. Hansen, and M. Packard. The nuclear induction experiment. *Phys. Rev.*, 70(7-8):474–485, Oct 1946.

[8] F. Breit. Magnetresonanztomographie - Was ist das eigentlich? In *Newsletter der FGF e.V.*, pages 19–25. Forschungsgemeinschaft Funk e.V., April 2006.

[9] F. Breuer. *Development and Application of Efficient Strategies for Parallel Magnetic Resonance Imaging*. PhD thesis, Julius-Maximilians-Universität Würzburg, Dec 2006.

[10] F. A. Breuer, P. Kellman, M. A. Griswold, and P. M. Jakob. Dynamic autocalibrated parallel imaging using temporal GRAPPA (TGRAPPA). *Magnetic Resonance in Medicine*, 53(4):981–985, 2005.

[11] I. Buck and T. Foley. BrookGPU. Stanford University Graphics Lab, 2003. http://graphics.stanford.edu/projects/brookgpu/, last accessed 2008/02/07.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[13] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM.

[14] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[15] T. C. Farrar and E. D. Becker. *Pulse and Fourier Transform NMR: Introduction to Theory and Methods*. Academic Press, 1971.

[16] D. Göddeke. GPGPU::basic math tutorial, 2005. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html, last accessed 2008/01/10.

[17] G. H. Golub and C. F. V. Loan. *Matrix computations*. Johns Hopkins University Press, Baltimore, MD, USA, 3rd edition, 1996.

[18] T. A. Gould. How MRI works, 2003. http://www.howstuffworks.com/mri.htm, last accessed 2007/12/30.

[19] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2003.

[20] R. Grimm. GPU-accelerated GRAPPA reconstruction in magnetic resonance imaging. Master thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, Department of Computer Science 9 (Computer Graphics), March 2008.

[21] M. A. Griswold, M. Blaimer, F. Breuer, R. M. Heidemann, M. Müller, and P. M. Jakob. Parallel magnetic resonance imaging using the GRAPPA operator formalism. *Magnetic Resonance in Medicine*, 54(6):1553–1556, Dec 2005.

[22] M. A. Griswold, P. M. Jakob, R. M. Heidemann, and M. Nittka. Generalized autocalibrating partially parallel acquisitions (GRAPPA). *Magnetic Resonance in Medicine*, 47:1202–1210, 2002.

[23] M. A. Griswold, P. M. Jakob, M. Nittka, J. W. Goldfarb, and A. Haase. Partially parallel imaging with localized sensitivities (PILS). *Magnetic Resonance in Medicine*, 44(4):602–609, 2000.

[24] M. Harris. General-purpose computation using graphics hardware, Nov 2002. http://www.gpgpu.org/, last accessed 2008/03/23.

[25] M. Harris. Tutorial S05: High performance computing on GPUs with CUDA, Session 05: Optimizing CUDA. In *Supercomputing 2007*. NVIDIA Research, 2007. http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf.

[26] R. M. Heidemann, M. A. Griswold, A. Haase, and P. M. Jakob. VD-AUTO-SMASH imaging. *Magnetic Resonance in Medicine*, 45(6):1066–1074, 2001.

[27] R. M. Heidemann, M. A. Griswold, M. Müller, F. Breuer, M. Blaimer, B. Kiefer, M. Schmitt, and P. M. Jakob. Möglichkeiten und Grenzen der parallelen MRT im Hochfeld. *Radiologe*, 44(1):49–55, January 2004. Feasibilities and limitations of high field parallel MRI.

[28] R. M. Heidemann, O. Ozsarlak, P. M. Parizel, J. Michiels, B. Kiefer, V. Jellus, M. Müller, F. Breuer, M. Blaimer, M. A. Griswold, and P. M. Jakob. A brief review of parallel magnetic resonance imaging. *Eur Radiol*, 13(10):2323–2337, Oct 2003.

[29] A. Hendrix. *Magnets, Spins and Resonances*. Siemens AG Medical Solutions, Magnetic Resonance, 2003.

[30] D. Hoa and A. Micheau. e-MRI, magnetic resonance imaging physics and technique course on the web. Campus Medica, 2007.
http://www.e-mri.org/, last accessed 2008/01/12.

[31] W. Hoge and D. Brooks. On the complimentarity of SENSE and GRAPPA in parallel MR imaging. In D. Brooks, editor, *Proc. 28th Annual International Conference of the IEEE Engineering in Medicine and Biology Society EMBS '06*, pages 755–758, 2006.

[32] J. Hornak. The basics of MRI, May 1996.
http://www.cis.rit.edu/htbooks/mri/, last accessed 2008/01/05.

[33] M. Houston. Tutorial S07: GPGPU: General-purpose computing on graphics hardware, Session 07: High level languages for GPUs. In *Supercomputing 2006*. Stanford University, 2006.
http://www.gpgpu.org/sc2006/slides/07.houston-high-level-languages.pdf.

[34] F. Huang, J. Akao, S. Vijayakumar, G. R. Duensing, and M. Limkeman. *k-t* GRAPPA: A *k*-space implementation for dynamic MRI with high reduction factor. *Magnetic Resonance in Medicine*, 54(5):1172–1184, 2005.

[35] D. Huo, D. Huo, and D. Wilson. Robust GRAPPA reconstruction. In D. Wilson, editor, *Proc. 3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro*, pages 37–40, 2006.

[36] Intel Corporation. Intel C++ compiler for linux, 2008.
http://www.intel.com/support/performancetools/c/linux/, last accessed 2008/03/02.

[37] Intel Corporation. Intel math kernel library 10.0, 2008.
http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm, last accessed 2008/03/02.

[38] Intel Corporation. PCI express x16 graphics interface, 2008.
http://www.intel.com/design/chipsets/pciexpress.htm, last accessed 2008/02/11.

[39] P. Jakob, M. Grisowld, R. Edelman, and D. Sodickson. AUTO-SMASH: A self-calibrating technique for SMASH imaging. *Magnetic Resonance Materials in Physics, Biology and Medicine*, 7(1):42–54, Nov 1998.

[40] M. Jiang and G. Wang. Convergence studies on iterative algorithms for image reconstruction. *IEEE Trans. Med. Imaging*, 22(5):569–579, 2003.

[41] A. C. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.
http://www.slaney.org/pct/, last accessed 2008/03/05.

[42] W. E. Kyriakos, L. P. Panych, D. F. Kacher, C.-F. Westin, S. M. Bao, R. V. Mulkern, and F. A. Jolesz. Sensitivity profiles from an array of coils for encoding and reconstruction in parallel (space rip). *Magnetic Resonance in Medicine*, 44(2):301–308, 2000.

[43] D. Lübke. Tutorial S05: High performance computing on GPUs with CUDA, Session 01: The democratization of parallel computing. In *Supercomputing 2007*. NVIDIA Research, 2007.
http://www.gpgpu.org/sc2007/SC07_CUDA_1_Introduction_Luebke.pdf.

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

[45] D. Manocha. General-purpose computations using graphics processors. *Computer*, 38(8):85–88, Aug 2005.

[46] D. W. McRobbie, E. A. Moore, M. J. Graves, and M. R. Prince. *MRI from picture to proton*. Cambridge University Press, January 2003.

[47] Microsoft Corporation. DirectX, 2008.
http://msdn.microsoft.com/directx/, last accessed 2008/02/07.

[48] Microsoft Corporation. Microsoft research accelerator project, 2008.
http://research.microsoft.com/research/downloads/Details/
25e1bea3-142e-4694-bde5-f0d44f9d8709/Details.aspx, last accessed 2008/02/07.

[49] H. Morneburg, editor. *Bildgebende Systeme für die medizinische Diagnostik*. Publicis MCD, Erlangen, 3rd edition, 1995. Röntgendiagnostik und Angiographie, Computertomographie, Nuklearmedizin, Magnetresonanztomographie, Sonographie, integrierte Informationssysteme.

[50] K. Müller, F. Xu, and N. Neophytou. Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography? *SPIE Electronic Imaging*, 2007.

[51] K. Müller and R. Yagel. Rapid 3-D cone-beam reconstruction with the simultaneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware. *Medical Imaging, IEEE Transactions on*, 19(12):1227–1237, Dec 2000.

[52] Nobel Web AB. The nobel prize in physics 1952, 2008. http://nobelprize.org/nobel_prizes/physics/laureates/1952/, last accessed 2007/12/30.

[53] Nobel Web AB. The nobel prize in physiology or medicine 2003, 2008. http://nobelprize.org/nobel_prizes/medicine/laureates/2003/, last accessed 2007/12/30.

[54] NVIDIA Corporation. Cg, 2003. http://developer.nvidia.com/page/cg_main.html, last accessed 2008/02/07.

[55] NVIDIA Corporation. *The CUDA Compiler Driver NVCC*, 1.1 edition, Octiber 2007. http://www.nvidia.com/object/cuda_get.htm, last accessed 2008/02/12.

[56] NVIDIA Corporation. *CUDA CUBLAS Library*, version 1.1 edition, September 2007. http://developer.download.nvidia.com/compute/cuda/1_1/CUBLAS_Library_1.1.pdf.

[57] NVIDIA Corporation. *CUDA CUFFT Library*, version 1.1 edition, October 2007. http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf.

[58] NVIDIA Corporation. *CUDA GPU Occupancy Calculator*, 2007. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[59] NVIDIA Corporation. *GPU Gems 3*. Addison-Wesley Professional, August 2007.

[60] NVIDIA Corporation. *NVIDIA CUDA - PTX: Parallel Thread Execution*, ISA version 1.1 edition, October 2007. http://www.nvidia.com/object/cuda_get.htm, last accessed 2008/02/11.

[61] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, version 1.1 edition, November 2007.
http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.

[62] NVIDIA Corporation. CUBLAS and CUFFT sources, February 2008.
http://forums.nvidia.com/index.php?showtopic=59101.

[63] NVIDIA Corporation. CUDA zone, 2008.
http://www.nvidia.com/object/cuda_home.html, last accessed 2008/02/07.

[64] NVIDIA Corporation. NVIDIA CUDA visual profiler, version 0.1 beta, February 2008.
http://forums.nvidia.com/index.php?showtopic=57443, last accessed 2008/02/11.

[65] OpenGL.org. OpenGL - the industry's foundation for high performance graphics, 1997.
http://www.opengl.org, last accessed 2008/02/07.

[66] OpenMP Architecture Review Board. The OpenMP specification for parallel programming, 2007.
http://www.openmp.org/, last accessed 2008/03/02.

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

[68] V. Pande. Folding@Home - distributed computing, 2007.
http://folding.stanford.edu/, last accessed 2008/02/12.

[69] W. Pauli. Exclusion principle and quantum mechanics - nobel lecture, December 13 1946.
http://nobelprize.org/nobel_prizes/physics/laureates/1945/pauli-lecture.pdf.

[70] K. P. Pruessmann, M. Weiger, P. Boernert, and P. Boesiger. Spiral SENSE: Sensitivity encoding with arbitrary k-space trajectories. In *ISMRM Conference Abstracts*, page 94, 1999.

[71] K. P. Prüssmann, M. Weiger, M. B. Scheidegger, and P. Bösiger. SENSE: Sensitivity encoding for fast MRI. *Magnetic Resonance in Medicine*, 42(5):952–962, 1999.

[72] E. M. Purcell, H. C. Torrey, and R. V. Pound. Resonance absorption by nuclear magnetic moments in a solid. *Phys. Rev.*, 69(1-2):37–38, Jan 1946.

[73] Rapid Mind, Inc. Rapid mind.
http://www.rapidmind.net/, last accessed 2008/02/07.

[74] H. Scherl, M. Koerner, H. Hofmann, W. Eckert, M. Kowarschik, and J. Hornegger. Implementation of the FDK algorithm for cone-beam CT on the cell broadband engine architecture. *Medical Imaging 2007: Physics of Medical Imaging*, 6510(1):651058, 2007.

[75] H. Scherl, H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-based CT reconstruction using the common unified device architecture (CUDA). In *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, volume 6, pages 4464–4466, 2007.

[76] T. Schiwietz, T. Chang, P. Speier, and R. Westermann. MR image reconstruction using the GPU. In *Proceedings of SPIE Medical Imaging 2006*, San Diego, CA, February 2006. SPIE.

[77] D. K. Sodickson and W. J. Manning. Simultaneous acquisition of spatial harmonics (SMASH): Fast imaging with radiofrequency coil arrays. *Magnetic Resonance in Medicine*, 38(4):591–603, 1997.

[78] S. Stone, H. Yi, W. mei Hwu, J. Haldar, B. Sutton, and Z.-P. Liang. How GPUs can improve the quality of magnetic resonance imaging. October 2007. The First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU), October 2007. Boston, MA.

[79] T. Sumanaweera and D. Liu. Medical image reconstruction with the FFT. In M. Pharr and R. Fernando, editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 765–784, Amsterdam, March 2005. Addison-Wesley Longman.

[80] J. Wang, T. Kluge, and M. Nittka. Parallel acquisition techniques with modified SENSE reconstruction mSENSE. In *Proceedings of the First international Workshop on parallel MRI basics and clinical applications*, page 398, Würzburg, 2001.

[81] X. Xue, A. Cheryauk, and D. Tubbs. Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: A simulation study. *SPIE Medical Imaging*, 2006.

[82] C. Zeller and M. Harris. Course 24: General-purpose computation on graphics hardware, Session 09: CUDA performance. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, New York, NY, USA, 2007. ACM.
http://www.gpgpu.org/s2007/slides/09-CUDA-performance.pdf.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, den 01. April 2008

Matthias Schneider