Title
Parallelism, Patterns, and Performance in Iterative MRI Reconstruction

Author
Murphy, Mark

Publication Date
2011

Peer reviewed|Thesis/dissertation

Parallelism, Patterns, and Performance in Iterative MRI Reconstruction

by

Mark Murphy

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael Lustig, Chair
Professor Kurt Keutzer, Chair
Professor Sara McMains

Fall 2011

Parallelism, Patterns, and Performance in Iterative MRI Reconstruction

Abstract

Parallelism, Patterns, and Performance in Iterative MRI Reconstruction

by

Mark Murphy

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Michael Lustig, Professor Kurt Keutzer, Chairs

Magnetic Resonance Imaging (MRI) is a non-invasive and highly flexible medical imaging modality that does not expose patients ionizing radiation. MR Image acquisitions can be designed by varying a large number of contrast-generation parameters, and many clinical diagnostic applications exist. However, imaging speed is a fundamental limitation to many potential applications. Traditionally, MRI data have been collected at Nyquist sampling rates to produce alias-free images. However, many recent scan acceleration techniques produce sub-Nyquist samplings. For example, Parallel Imaging is a well-established acceleration technique that receives the MR signal simultaneously from multiple receive channels. Compressed sensing leverages randomized undersampling and the compressibility (e.g. via Wavelet transforms or Total-Variation) of medical images to allow more aggressive undersampling. Reconstruction of clinically viable images from these highly accelerated acquisitions requires powerful, usually iterative algorithms. Non-Cartesian pulse sequences that perform non-equispaced sampling of k-space further increase computational intensity of reconstruction, as they preclude direct use of the Fast Fourier Transform (FFT). Most iterative algorithms can be understood by considering the MRI reconstruction as an inverse problem, where measurements of un-observable parameters are made via an observation function that models the acquisition process. Traditional direct reconstruction methods attempt to invert this observation function, whereas iterative methods require its repeated computation and computation of its adjoint. As a result, naïve sequential implementations of iterative reconstructions produce unfeasibly long runtimes. Their computational intensity is a substantial barrier to their adoption in clinical MRI practice.

A powerful new family of massively parallel microprocessor architectures has emerged simultaneously with the development of these new reconstruction techniques. Due to fundamental limitations in silicon fabrication technology, sequential microprocessors reached the power-dissipation limits of commodity cooling systems in the early 2000's. The techniques used by processor architects to extract instruction-level parallelism from sequential programs face ever-diminishing returns, and further performance improvement of sequential processors

via increasing clock-frequency has become impractical. However, circuit density and process feature sizes still improve at Moore's Law rates. With every generation of silicon fabrication technology, a larger number of transistors are available to system architects. Consequently, all microprocessor vendors now exclusively produce multi-core parallel processors. Additionally, the move towards on-chip parallelism has allowed processor architects a larger degree of freedom in the design of multi-threaded pipelines and memory hierarchies. Many of the inefficiencies inherent in superscalar out-of-order design are being replaced by the high efficiency afforded by throughput-oriented designs.

The move towards on-chip parallelism has resulted in a vast increase in the amount of computational power available in commodity systems. However, this move has also shifted the burden of computational performance towards software developers. In particular, the highly efficient implementation of MRI reconstructions on these systems requires manual parallelization and optimization. Thus, while ubiquitous parallelism provides a solution to the computational intensity of iterative MRI reconstructions, it also poses a substantial software productivity challenge.

In this thesis, we propose that a principled approach to the design and implementation of reconstruction algorithms can ameliorate this software productivity issue. We draw much inspiration from developments in the field of computational science, which has faced similar parallelization and software development challenges for several decades. We propose a Software Architecture for the implementation of reconstruction algorithms, which composes two Design Patterns that originated in the domain of massively parallel scientific computing. This architecture allows for the most computationally intense operations performed by MRI reconstructions to be implemented as re-usable libraries. Thus the software development effort required to produce highly efficient and heavily optimized implementations of these operations can be amortized over many different reconstruction systems. Additionally, the architecture prescribes several different strategies for mapping reconstruction algorithms onto parallel processors, easing the burden of parallelization. We describe the implementation of a complete reconstruction, ℓ_1-SPIRiT, according to these strategies. ℓ_1-SPIRiT is a general reconstruction framework that seamlessly integrates all three of the scan acceleration techniques mentioned above. Our implementation achieves substantial performance improvement over baseline, and has enabled substantial clinical evaluation of its approach to combining Parallel Imaging and Compressive Sensing. Additionally, we include an in-depth description of the performance optimization of the non-uniform Fast Fourier Transform (nuFFT), an operation used in all non-Cartesian reconstructions. This discussion complements well our description of ℓ_1-SPIRiT, which we have only implemented for Cartesian samplings.

To Adrienne

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Magnetic Resonance Imaging (MRI) is a highly flexible and non-invasive medical imaging modality. Many medical imaging methods expose patients to ionizing radiation, such as Computed Tomography (CT), X-ray, and Positron Emission Tomography (PET). MRI relies entirely on Nuclear Magnetic Resonance (NMR), a quantum-physical phenomenon by which magnetically-polarized nuclei emit an electromagnetic signal. Producing an NMR signal from a patient's body requires exposure only to non-ionizing radio frequency electromagnetic radiation. Additionally, MR scans are highly parametrizable. The flexibility of the NMR phenomenon and the scanner hardware enables a vast array of clinical applications of MRI. Depending on the scan parameters and the pulse sequence, MR Image contrast can be sensitive to a number of physical properties, enabling extremely flexible imaging of soft-tissue. Furthermore, MR image contrast can be made sensitive to fluid flow velocities, diffusion, temperature, or blood oxygenation levels near active neurons. Consequently, MRI is a highly desirable imaging modality for many clinical applications.

Unfortunately, MRI is much slower and more costly than other modalities. As a result, MR accounts for only a small fraction of medical images in practice. MR systems' high cost is due to several factors. The electronics of the scanner are sensitive and complex, and liquid-helium cooled superconducting magnets produce the extremely high magnetic fields required for NMR. While the high cost of MR is one factor that limits its applicability in some contexts, its inherently slow data acquisition is also a substantial barrier to wider adoption. The slowness of acquisition makes it difficult to apply MRI to time-sensitive clinical applications. For example, time-resolved imaging is more difficult in 3D than in 2D, since volumetric scans are typically two orders of magnitude slower in both acquisition and reconstruction.

From a computer science perspective, the costs associated with the physical system are unavoidable and fixed. However, recent advances in sampling and reconstruction have substantially improved imaging speed. As we discuss in Chapter 2, a number of efficient sampling schemes have been proposed to reduce the amount of data acquired during a scan. As MR is able to sample the image point-by-point sequentially, a reduction in the amount of data

acquired translates directly to scan acceleration. The need for magnetization recovery between readouts results in low duty-cycle of acquisition, and scan acceleration techniques are crucial to expanding and improving the clinical applications of MRI. However, these techniques sample the image far below the Nyquist rate, and images produced via traditional reconstructions are dominated by noise and aliasing.

This thesis discusses the advanced reconstruction techniques required to produce clinical-quality images from highly accelerated scans and the software optimization techniques required to achieve clinically feasible reconstruction runtimes. Many of these recently proposed reconstructions are fundamentally iterative, as they leverage powerful signal processing, numerical optimization, and linear algebra algorithms. These reconstructions are able to remove aliasing from undersampling and regain lost signal-to-noise ratio (SNR). Despite promising results, their adoption is limited by the intense computation they require. Naïve implementations of these reconstructions run for hours, whereas an on-line clinical application permits only a few minutes of runtime. A crucial component of any reconstruction system is a fast implementation of the computationally intense reconstruction algorithm.

From the early 1980's to the early 2000's, little additional software development effort was required for the performance benefits of Moore's Law to be manifest in end-user applications. The rapid advance of sequential microprocessor architectures and compiler technologies accelerated most workloads by over 100× in this era. Over the past decade, however, the performance of general-purpose workloads on single-core processors has all but stalled. In recent years parallelism has become a ubiquitous component of general-purpose microprocessor designs. The term *manycore processors* has been used to describe these architectures. In prior generations of uniprocessor architectures, high clock rates and moderate degrees of instruction-level parallelism were the primary mechanism by which processors increased performance. In manycore processors, performance improvement from generation to generation is primarily provided by large-scale programmer-visible parallelism. For performance to scale with Moore's law, software must explicitly take advantage of the parallel execution and memory resources that manycore processors provide. This additional optimization effort is a substantial burden on application developers, few of whom posses the necessary expertise in computer architecture and low-level performance tuning. Furthermore, if near peak performance is to be achieved, then substantial re-optimization and re-parallelization may be necessary when migrating an application to a next-generation processor architecture. Consequently, it is impossible for the majority of application developers to remain entirely agnostic to all microarchitectural concerns. Fundamentally, the burden of performance is shifting from the processor architect to the application developer. These facts pose a serious software productivity problem: microprocessor architecture and silicon fabrication technologies alone are no longer capable of improving performance. Until compiler technology is able to automate the implementation decisions discussed in this thesis, substantial programmer intervention is the only way to increase computational performance.

In this thesis, we discuss the design of high-performance software systems for these computationally intense MRI reconstructions. Many recently proposed reconstructions can be

described via formulation as an inverse problem, where system parameters are to be estimated from measurements made via a system transfer function. Much of the computational intensity of iterative reconstructions lies in the application of the system transfer function and its adjoint operator. The system function models the MRI data acquisition process as a composition of primitive operations such as Fourier transforms and sensitivity encodings. The iterative algorithms that solve these inverse problem require repeated application of these operations, whereas traditional direct reconstructions typically compute them only once. The software that implements these algorithms must be developed primarily by MRI scientists and researchers, who do not necessarily have the prerequisite knowledge of computer architecture to perform low-level performance optimization. Thus it is crucial that the design of these software systems separate responsibility for these optimization issues from the design of the inverse-problem formulations and reconstruction algorithms.

We propose a novel software architecture, the *Supermarionation* architecture, which achieves this separation, and describe its application in the MRI reconstruction context. The models of MRI acquisition used in various reconstructions rely on a single, small set of primitive operations. Thus, they can be tuned for the important cases encountered in MRI reconstruction and re-used as a library. The Supermarionation architecture is a combination of several previously proposed Design Patterns. Patterns are a well-established technique of describing software design issues. These patterns, *Geometric Decomposition* [58] and *Puppeteer* [70], are drawn from the well-developed field of massively parallel scientific computing, which for decades has faced the same software productivity problems that we have described here. Our proposed software architecture separates the low-level tuning and optimization of the most performance-sensitive parts of the reconstruction system from the implementation of the iterative inverse-problem solver. However, it cannot hide all the implications of low-level optimization decisions, because the researcher implementing the reconstruction algorithm must be aware of the layout of data in memory. The data-partitioning scheme drives the parallelization of all computations performed by the reconstruction, even ones that cannot be performed by library functions and thus must be implemented by the MRI scientist.

1.1 Outline

The remainder of this thesis is organized as follows:

- Chapter 2 provides necessary background on magnetic resonance imaging, reconstruction algorithms and high-performance computing.
- Chapter 3 describes the *Supermarionation* architecture and its application in the design of high-performance MRI reconstructions. This chapter describes the elements of a reusable library to aid in the implementation of MRI reconstruction systems, and the design of iterative algorithms based on ths library.

- Chapter 4 is an in-depth description of a reconstruction system implemented according to the aforementioned strategies. This system uses several of the library components described in the previous chapter.
- Chapter 5 is an in-depth description of the optimization of one element of the library described above: the Non-Uniform Fast Fourier Transform, known commonly as Gridding in MRI reconstruction. In a complete library implementation, similar techniques could be used for all of the computationally intense library elements.
- Chapter 6 includes a summary, discussion, and conclusion.

Chapter 2

Background

2.1 Introduction

This chapter presents background material necessary to more completely understand the chapters which follow. The topic of this thesis is a hybrid between two fields, Magnetic Resonance Imaging (MRI) and High-Performance Computing (HPC). We present the background from these fields separately in Sections 2.2 and 2.3, respectively.

The scope of active research and clinical practice of MR imaging is far too broad for us to hope to discuss in depth this thesis. The extreme flexibility of the Nuclear Magnetic Resonance (NMR) phenomenon has produced many clinical diagnostic applications of the technique, each of which warrants a PhD dissertation in its own right. The design of efficient gradient and radio frequency pulse sequences for exciting and manipulating NMR spins is an equally complex and rich field of study, which we cannot hope to discuss in depth. The topic of this thesis is the computational performance of the algorithms used to produce images from the NMR signal. As such, the goal of Section 2.2 is to bridge the gap from a classical description of the physics underlying magnetic resonance to the algorithms that fall within our scope. The goal of Section 2.3 is to describe the current state-of-the-art in parallel computing systems, and to describe the difficulties of parallel software development and performance optimization that MRI reconstruction systems must face.

2.2 Magnetic Resonance Image Reconstruction

Although Nuclear Magnetic Resonance (NMR) is fundamentally a quantum-mechanical phenomenon, the behavior of atomic nuclei can be very well described using a simple classical model. The classical model is sufficient for the purposes of describing the algorithms used during image reconstruction, but we must take several quantum-mechanical properties axioms, for example the concepts of spin angular momentum and Larmor precession frequency. In clinical imaging, the NMR signal is produced primarily by the protons – i.e. ^{1}H atoms

– in water molecules in the patient's body. Other species present in the body are able to produce an NMR signal, but ^{1}H is by far the most abundant. Since they posses an odd number of nucleons, ^{1}H atoms exhibit the quantum spin angular momentum property. For this reason, ^{1}H atoms are commonly called *spins* in MRI parlance. Since protons also exhibit a +1 electrical charge, we can model them as spinning charged spheres. The combination of spin and charge results in a magnetic moment, of which MR imaging systems take advantage to produce the NMR signal.

2.2.1 Electromagnetic Fields used in MRI

In clinical imaging, the NMR signal is produced by applying three different electromagnetic fields to the spins in a patient's body. The first field is a strong, static field used to align the magnetic moments of the spins. This field is of constant magnitude and direction throughout the field of view (FOV). By convention, its direction is defined as the $\hat{z}$ axis, and it is called the $\mathbf{B}_0 = B_0\hat{z}$ field. In typical clinical imaging systems, B_0 is 1.5 to 3 Teslas – approximately 4 orders of magnitude stronger than the Earth's magnetic field. The second type of field used in MRI are short-lived time-varying radio frequency pulses, similar to the radiation used in FM radio transmission. These pulses are conventionally referred to as $\mathbf{B}_1$ fields. The $\mathbf{B}_1$ field's carrier frequency is tuned to the resonant Larmor frequency $\omega = \gamma B_0$, where γ is a constant whose value depends on the specie to be imaged. For the common case of ^{1}H imaging $\gamma = 267.52 \times 10^6$ radians per second per Tesla. The effect of the $\mathbf{B}_1$ pulse is to rotate the spins, perturbing their alignment to $\mathbf{B}_0$. The strength, duration, and polarization of $\mathbf{B}_1$ are chosen precisely to control this perturbation. After perturbation, spins tend to re-align with $\mathbf{B}_0$ at rates described by the T_1 and T_2 parameters which we'll discuss momentarily in terms of the Bloch equation. The final fields used in MR imaging are the time-varying Gradient fields $\mathbf{G}(t) = \left[\frac{\partial B_z}{\partial x}(t), \frac{\partial B_z}{\partial y}(t), \frac{\partial B_z}{\partial z}(t)\right]^T$, which vary the strength of the magnetic field parallel to the $\hat{z}$ axis. We have written $\mathbf{G}$ as a function of time only, rather than of both time and space, since in MRI the gradient field is almost always linear and constant throughout the FOV.

2.2.2 Bloch Equation

The Bloch Equation is the aforementioned classical model of the behavior of NMR spins in the presence of electromagnetic fields. If we denote the spatial position and time variables as $\mathbf{x}$ and t, the magnetization of spins in the FOV as $\mathbf{M}(\mathbf{x}, t) = (M_x, M_y, M_z)$, and the total magnetic field as $\mathbf{B}(\mathbf{x}, t) = (B_x, B_y, B_z)$, then the Bloch Equation is

$$\frac{d\mathbf{M}}{dt} = \mathbf{M} \times \gamma\mathbf{B} - \frac{M_x\hat{x} + M_y\hat{y}}{T_2} - \frac{(M_z - M_0)\hat{z}}{T_1} \tag{2.1}$$

where $\times$ denotes a vector cross product. The first term $\mathbf{M} \times \gamma\mathbf{B}$ describes the precession of spins about the $\hat{z}$ axis which ultimately is the source of the NMR signal. A familiar

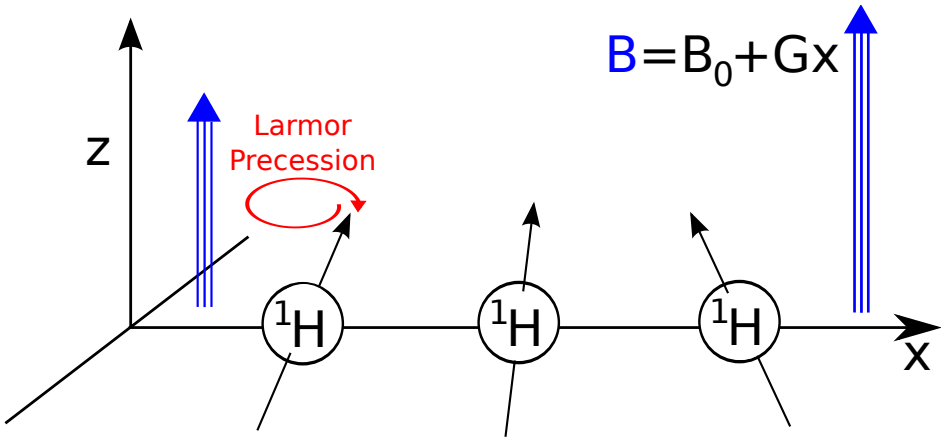

Figure 2.1: Illustration of nuclear magnetic resonance under the influence of the fields used in MRI. Three ^{1}H spins are shown with the orientation of their magnetic dipoles. Due to the gradient field $\mathbf{G}$, the strength of the magnetic field in the $\hat{z}$ direction varies across the field of view. Accordingly, the Larmor precession frequencies vary as well. In this diagram the field strength varies linearly, increasing with x, so that the precession frequencies $\omega(x) = \gamma B(x) = \gamma(B_0 + Gx)$. As the spins precess at different rates, they accrue phase with respect to each other.

analogy can be made to the child's toy known as a "top", which spins about a vertical axis and is held upright by its angular momentum. When perturbed from exact verticality, the force of Gravity causes precession via a similar cross-product force. The second and third terms of the Bloch Equation describe the T_2-*decay* of the transverse (i.e. in the $x-y$ plane) magnetization and the T_1-*recovery* of the longitudinal (i.e. parallel to $\hat{z}$) magnetization. The values of the T_1 and T_2 parameters vary according to the tissue in which the ^{1}H lie. The variance of these parameters across tissues is frequently used to generate image contrast. By inserting appropriate delays into the pulse sequence between $\mathbf{B}_1$ excitation and signal measurement, the strength of the signal is made to vary among the various tissues present in the FOV.

2.2.3 Signal Equation

The NMR signal is received via Faraday's Law, which states that a varying magnetic flux through a closed conductive loop induces current. MR systems place conductive loops nearby the sample containing spins whose magnetizations are evolving according to Equation 2.1. In our context, the conductive loops are MRI receiver coil, tuned to resonate signals at the Larmor frequency. As the spins precess, their magnetic fields oscillate. This oscillation produces the varying magnetic flux though the receiver coils, resulting in a measurable

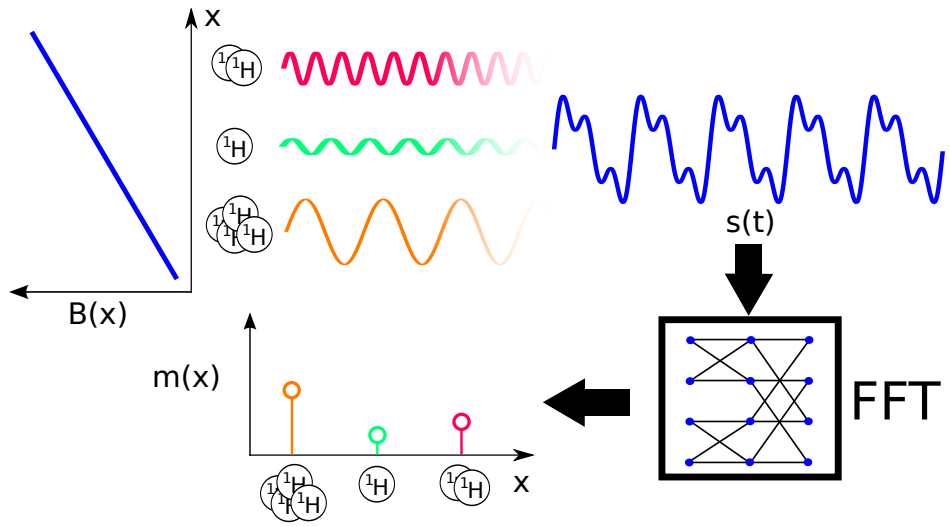

Figure 2.2: Demonstration of the Fourier relationship in Nuclear Magnetic Resonance (NMR) imaging. A spatial distribution of ^{1}H atoms emits a radio frequency signal after excitation via a radio frequency pulse. The field strength $B(x) = B_0 + Gx$ varies linearly across the field of view, and thus the frequency of the signal emitted by each ^{1}H atom depends linearly on its position via the Larmor relationship $\omega(x) = \gamma B(x)$. The received NMR signal is the sum of the signals emitted by all protons, but its spectral components can be recovered via a Fourier transform. Since the linear field strength variation produces a linear relationship between position and frequency, the Fourier transform recovers the spatial distribution of ^{1}H atoms.

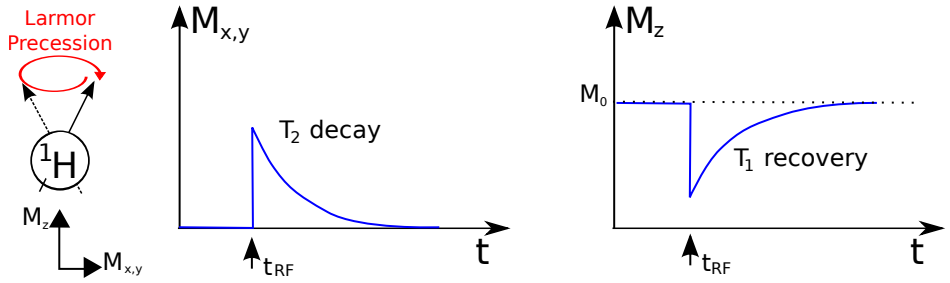

Figure 2.3: Illustration of T_2 decay and T_1 recovery. The magnetization vector $\mathbf{M}$ for a given spin resolves into the $\hat{x}$, $\hat{y}$, and $\hat{z}$ components: $\mathbf{M} = (M_x, M_y, M_z)'$. We denote the magnitude of the transverse magnetization $M_{x,y} = ||M_x\hat{x} + M_y\hat{y}||$. Precession is about the $\hat{z}$ axis only. The M_x and M_y components vary sinusoidally, but both $M_{x,y}$ and M_z vary smoothly according to the decay and recovery rates. We assume that at $t = 0$, the spin's magnetization is aligned with the $\mathbf{B}_0$ field along the $\hat{z}$ axis. At some time $t_{\text{RF}} > 0$ the radiofrequency $\mathbf{B}_1$ pulse rotates the magnetization away from the $\hat{z}$ axis, causing the transverse magnetization $M_{x,y}$ to increase and the longitudinal magnetization M_z to decrease. After t_{RF}, the transverse magnetization $M_{x,y}$ decays exponentially at the T_2 rate and the longitudinal magnetization recovers exponentially at the T_1 rate.

voltage. Since the signal is received simultaneously for all spins in the FOV, we write the signal as the integral

$$s(t) = \int_{\text{FOV}} M_{xy}(\mathbf{x}, t)d\mathbf{x}.$$

Inserting a solution to the Bloch Equation[1] into this integral produces the MRI Signal Equation:

$$s(t) = \int_{\text{FOV}} M_{xy}(\mathbf{x})e^{-t/T_2(x)}e^{-\imath\omega_0 t}\exp\left(-\imath\gamma\int_0^t \langle\mathbf{x}, \mathbf{G}(\tau)\rangle\, d\tau\right)d\mathbf{x} \tag{2.2}$$

where $\langle\cdot,\cdot\rangle$ denotes inner (dot) product. Equation 2.2 is rather unwieldy, and it is appropriate to make several simplifications. First, we can ignore the carrier-frequency term $e^{-\imath\omega_0 t}$, as the NMR signal is demodulated at the Larmor frequency ω_0 before processing. Second, the length of time during which data is acquired is usually much shorter than the value of T_2, and in most cases the T_2 decay term $e^{-t/T_2(x)}$ can be ignored. Third, we define the *k-space* coordinates to be $\mathbf{k}(t) = \frac{\gamma}{2\pi}\int_0^t \mathbf{G}(\tau)d\tau$. With these simplifications and the appropriate algebraic manipulation, the signal equation reduces to

$$s(t) = \int M_{xy}(\mathbf{x})e^{-2\pi\imath\langle\mathbf{k}(t),\mathbf{x}\rangle}d\mathbf{x} \tag{2.3}$$

[1]Derivation of the Bloch Equation's solution is beyond our scope

From Equation 2.3 it is clear that the received time-domain signal is equivalent to a Fourier transform of the transverse magnetization evaluated at spatial frequencies $\mathbf{k}(t)$, which are determined by the Gradient pulse sequences applied during signal acquisition. Thus, the task of reconstructing an image of the transverse magnetization M_{xy} from the samples of $s(t)$ is equivalent to computing an inverse Fourier transform.

The Fourier interpretation of the signal equation was a crucial development in the theory of MRI. In practice the received time-domain signal $s(t)$ is sampled at a discrete set of time-points $\{\bar{t}_i\}$, which correspond to a set of spatial-frequencies $\{\mathbf{k}(\bar{t}_i)\}$. The most commonly used pulse sequence designs choose an equally-spaced set of sample locations $\{\mathbf{k}(\bar{t}_i)\}$ that satisfy the Nyquist-Shannon sampling criteria for a desired image resolution and field-of-view. The image can then be reconstructed very rapidly via the well-known Fast Fourier Transform (FFT) algorithm. This acquisition technique is known as 2DFT or 3DFT, for 2D and 3D imaging respectively, and is the standard clinical imaging practice for most applications.

2.2.4 MRI Reconstructions as Inverse Problems

In general, an *Inverse Problem* is any situation in which measurable quantities are related to parameters of a system by some known (or theorized) relationship. One cannot directly observe the values of the parameters but wishes to estimate them from the measurements. We denote the values measured as $y \in \mathbb{C}^m$, the parameters to be estimated as $x \in \mathbb{C}^n$, and the function that computes the relationship as $F(x)$. In inverse problems, one has observed some measurements y of a system represented by $F(\cdot)$, and desires to compute the values x so that $y = F(x)$. This is a very general framework that describes a wide variety of scientific and engineering problems, and it has been studied extensively. Depending on the particular properties of the system function $F(\cdot)$, a variety of very powerful methods exist for estimating the parameters x.

In the above discussion of MRI physics, we described the acquired signal as a Fourier transform of the transverse magnetization. To view this as an inverse problem, we can define the parameters x to be estimated as a discretization of the transverse magnetization, and the observed measurements y to be the samples of the NMR signal. The system function F is then a linear operator that evaluates the (discrete) Fourier transform of x at the spatial frequencies corresponding to the Gradient pulse sequence. In the common case of equispaced Nyquist-Shannon sampling, the system function is simply the square Discrete Fourier Transform matrix, which we denote $\mathcal{F}$. Since this matrix is square and full-rank, the reconstruction can be be performed via its inverse: $x = \mathcal{F}^{-1}y$. As noted above, this matrix-vector product can be computed efficiently in $O(n \log n)$ time via the FFT.

The most common approach to accelerating MRI data acquisition is to acquire equispaced k-space locations, but to sample one or more dimensions below the Nyquist rate. The system matrix for the inverse problem formulation of this approach consists of a subset of the rows of the DFT matrix, which we denote as $\mathcal{F}_u \in \mathbb{C}^{m \times n}$, with $m < n$. The inverse problem in this case is the under-determined linear system $y = \mathcal{F}_u x$, which has no unique solution x.

The reconstruction must incorporate further knowledge of the system in order to resolve the ambiguity.

In most clinical applications of MRI, the signal is acquired simultaneously from multiple receiver coils. Typically 4-32 coils are used. This practice, known as Parallel Imaging, allows the coils to be placed much closer to the patient's body. Each coil will receive the NMR signal with substantially higher signal-to-noise ratio (SNR), but with a spatial weighting. The signal will be received much more strongly from spins nearby the coil. The effect on the signal is partially due to electromagnetic coupling between the receiver coil and the patient's body, and it must be estimated separately for each patient. In the inverse-problem formalism, we can model this spatial weighting as multiplication of the signal x by a diagonal matrix S_i for each of the c coils. The inverse problem becomes

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_c \end{pmatrix} = \begin{bmatrix} \mathcal{F}_u \cdot S_1 \\ \vdots \\ \mathcal{F}_u \cdot S_c \end{bmatrix} x \tag{2.4}$$

where y_i is the signal received from the i^{th} receiver coil, and the matrix in square brackets is the system matrix, composed of the diagonal matrices S_i representing the coil spatial sensitivities. The original SENSE [67] reconstruction solved this inverse problem by exploiting the aliasing pattern that arises in regularly-subsampled imaging. Iterative SENSE [53] reconstructions have solved this problem via numerical linear algebra algorithms [5] such as Conjugate Gradients. These approaches typically assume that the sensitivities S_i are known explicitly or can be accurately estimated, for example from a prescan. *Auto-calibrating* over-sample the center of k-space in order to re-estimate the sensitivies for each scan. The approaches of Ying *et al.* [90] and Uecker *et al.* [79] attempt to estimate the sensitivities simultaneously with the estimation of the data x. This reconstruction problem is non-linear, and Uecker *et al.* use an iteratively re-weighted Gauss-Newton algorithm. In general the sensitivities are difficult to estimate accurately and reliably, a fact which motivates reconstructions that use sensitivity information implicitly. GRAPPA [37, 57] is an alternate approach to Parallel Imaging reconstruction does not require an explicit representation of the sensitivities, instead estimating the sensitivity-weighted images $x_i = S_i x$ via linear algorithms.

Another approach to resolving the ambiguity arising from the under-determinedness of sub-sampled reconstructions is motivated by the ℓ_0-ℓ_1 equivalence result of Compressive Sensing [16, 26, 55]. Compressive Sensing relies on the existence of a sparse representation of the data x via some change of basis $\alpha = \Psi x$. When such a sparse representation exists, the signal x can be reconstructed from sub-sampled linear measurements $y = \mathcal{F}_u x$ by minimizing the convex ℓ_1 norm of α. The ℓ_1 norm is defined as the sum of absolute values (complex moduli) of the components of a vector: $||x||_1 = \sum_i |x_i|$. I.e. Compressive Sensing theory prescribes that the desired image x is the solution of the numerical optimization:

$$\begin{aligned} \min_x \quad & ||\Psi x||_1 \\ \text{s.t.} \quad & y = \mathcal{F}_u x \end{aligned}$$

This problem can be reformulated as a Linear Program (LP), but the methods used to solve LP generally do not scale to size of problems encountered in MRI reconstruction. However, a number of highly scalable first-order algorithms have recently been proposed [19, 22, 8, 77, 25, 29, 86].

The final scan acceleration approach that we discuss is non-Cartesian sampling of k-space. Until now, we've assumed that $\mathcal{F}_u$ is related to the DFT matrix $\mathcal{F}$ via multiplication by a subset of the Identity matrix: $\mathcal{F}_u = I_u\mathcal{F}$. $\mathcal{F}_u x$ and $\mathcal{F}_u^{-1} y$ can both be computed rapidly via the FFT. The discrete Fourier transform (DFT) can still be represented as an $m \times n$ matrix, but the factorization of the DFT matrix that produces the FFT requires that the sample locations be equispaced. Non-Cartesian pulse sequences remove the restriction that k-space samples be equispaced, and instead acquire samples along trajectories $\mathbf{k}(t)$ that are particularly convenient to generate with MRI gradient-generation hardware. As a result data can be acquired much more rapidly, even without undersampling. However, the inverse Fourier transform of the non-equispaced samples cannot be computed via the FFT. Computation of the DFT directly requires $O(mn)$ time and is unfeasibly long for many MRI applications, however Chapter 5 of this thesis describes in detail the fast algorithm by which MRI reconstructions approximate the non-equispaced DFT.

2.3 High Performance Computing

2.3.1 Asymptotic Analyses

Most scientists and engineers leveraging computational methods are familiar with the notion of asymptotic complexity, commonly called "Big-Oh" notation. The statement

$$f(n) \in O\left(g(n)\right)$$

is equivalent to the statement that there exists some constant C such that in the limit as $n \to \infty$, the inequality

$$|f(n)| \leq C|g(n)|$$

holds. Asymptotic complexity is a crucial tool in performance analysis, as it provides a relative measure of the runtime of algorithms. Typically, one identifies a polynomial expression for the number of times an algorithm will perform some particular operation. All but the terms with the largest exponent can be ignored for the purposes of asymptotic analysis. Although asymptotics are crucial to understanding performance of an algorithm, more analysis of the constant factors and lower-level architectural issues are necessary to guide performance optimization decisions.

For example, consider the multiplication of two matrices A and B stored in column-major order contiguously in memory. This operation is sometimes referred to as `GEMM`, short for GEneral Matrix-Matrix multiplication, a name drawn from the Basic Linear Algebra

Subroutines (BLAS) library as originally implemented in Fortran. Each entry $c_{i,j}$ of the product $C = AB$ is computed as the inner (dot) product of the i^{th} row of A with the j^{th} column of B. If the matrices involved are all $n \times n$, then the matrix product performs $2n^3$ floating-point operations. The naïve implementation of matrix-matrix multiplication consists of three nested loops: over rows of A, columns of B, and over the dot product sum between a given row/column pair. All implementations have the same $O(n^3)$ asymptotic complexity, but this straightforward implementation can run over $1,000\times$ slower for large matrices than the implementations provided in high-performance linear algebra libraries. The reasons for this performance disparity are the low-level optimizations applied to the high-performance library. These low-level optimizations are intended to exploit the low-level features of the architecures we'll describe in Section 2.3.3.

2.3.2 "Laws" of computing

There are several fundamental facts commonly referred to as "Laws" in computing parlance, and are pertinent to our study of performance. The first, Moore's Law, describes the evolution of silicon fabrication technology over time. The second, Amdahl's Law, describes the effect of performance optimization for a single step of a multi-step algorithm. The third, Gustafson's Law, describes the scaling of performance with increasing input size.

Moore's Law [59] is an empirical observation of the rapid pace of improvement in the process technology by which microprocessors are manufactured. Moore's original observation was made based on only a few successive technology nodes in 1965, but it has held true nearly to this day: the number of transistors per unit area of integrated circuit has doubled approximately every two years for the past 40 years. In the present day, the trend of rapidly increasing circuit density continues. State-of-the-art microprocessors in 2011 integrate up to 3 billion transistors in approximately 500 mm^2 of silicon die. In the past 5 to 10 years, the rapid increase in circuit density has reached a fundamental limitation in power dissipation. In the 1990's and early 2000's, increases in circuit density were used to increase pipeline depths and accompanied by increases in clock frequency beyond what normal technology scaling can provide. Uniprocessor performance improved correspondingly. Dynamic power dissipation increases linearly with clock frequency, and eventually power dissipation reached the limits of the cooling systems used in commodity computer systems. Consequently, the microprocessor industry now uses the Moore's Law increase in circuit density to integrate larger number of processor cores per die. Uniprocessor performance has stalled, but total computational throughput continues to improve. A crucial result of this shift towards on-chip parallelism is the growing importance of the memory system in performance-optimization. While the circuit-density of microprocessors continues to improve, the area and diameter of physical microprocessor packages remains approximately the same. The number of instructions-per-second a processor can execute is roughly proportional to the number of transistors (i.e. cores) on chip. DRAM is fabricated via different processes and must reside in separate packages, and the rate at which data can transfer between processors and memory is limited

by these physical packages. Thus memory-system performance is increasingly a limiting factor in many numerical and scientific computing applications.

Amdahl's Law [3] is the observation that most computations perform several different sub-computations, and overall performance gain from improving a single sub-computation is limited. For example, suppose that a particular computation consumes half of the runtime of an algorithm. If one can improve the performance of that computation by $1,000\times$, then one has only improved the runtime of the algorithm by $2\times$. In general, the performance improvement from optimization (e.g., parallelization) is limited by the fraction of total runtime the optimized computation represents. If we denote that fraction as p, the performance improvement from optimization as s, and the new and original total runtimes as r' and r, then Amdahl's law can be stated as

$$r' = r\left((1-p) + \frac{p}{s}\right)$$

As the speedup s increases towards $+\infty$, the relative runtime approaches $r(1-p)$. An oft-quoted[2] corollary to Amdahl's law cautions against optimizing the wrong computations: "Premature optimization is the root of all evil." Performance optimization can be a very labor-intensive process, consisting of much trial and error. It is undesirable to perform any optimization until one knows where the performance bottlenecks are.

The final law, Gustafson's Law, applies in many numerical and scientific computing applications. Gustafson [39] noted that in many cases, the quality of the result produced by a computation can be improved by increasing the size of the input. For example in a fluid-dynamics simulation, the accuracy of the computed result will be higher if a finer-scale discretization of the volume is used. The finer discretization corresponds to a much larger input size. Increasing the input size will lengthen runtime, but Gustafson noted that the asymptotically more expensive computations in an algorithm will come to dominate runtime. If these asymptotically more expensive operations are easier to optimize or parallelize than the other computations, then one should certainly target them for performance improvement before considering the other operations. Although in MRI reconstruction the size of input is fundamentally limited by the amount of time the radiologist can afford to spend acquiring data, Gustafson's law provides a powerful insight: in deciding where to spend precious programmer effort in performance optimization, it is crucial to first benchmark the reconstruction algorithm for realistic problem sizes. Otherwise, our optimization efforts can easily be misdirected at unimportant computations.

2.3.3 Modern Processor Architectures

There are two important components to modern processor architectures: memory systems and instruction execution hardware. Much discussion of performance optimization for modern processing systems can be divided according to whether a particular optimization effects

[2] Attribution is disputed, and the truism is due either to Tony Hoare or Donald Knuth

instruction-execution performance, or whether it effects memory-system performance. As mentioned above, the trend of on-chip parallelism leads us to believe that most performance optimizations will focus on improving memory-system performance. However, much of the recent innovation in processor architecture that has stirred interest in high-performance parallel programming lies in the mechanisms by which parallelism is exposed to the programmer. From a historical perspective, this resurgence in interest began when Graphics Processing Units (GPUs) began to incorporate programmable processor cores. For much of the 1990s, GPUs included only fixed-function hardware for performing the projective geometry and texture interpolation required for rendering 3-D graphics. As graphics algorithms became more diverse, however, GPU vendors began to integrate programmable cores into their fixed-function graphics rendering pipelines. To this day, much of the die-area of GPUs is devoted to non-programmable fixed-function circuitry (e.g. texture interpolation), but the programmable cores execute Turing-complete instruction sets with the ability to read and write arbitrary locations in memory. Note that much of the die-area of "general-purpose" CPU processors is devoted to the out-of-order instruction execution hardware intended to optimize the sequential programs represented by the SPEC [40] benchmarks, which have helped to drive CPU design for over two decades. Arguably, neither the texture-interpolation functional units nor the out-of-order instruction execution circuitry are strictly necessary for the efficient implementation of scientific and numerical software like MRI reconstructions, so neither type of processor is at a clear disadvantage.

Many of the same micro-architectural mechanisms are used in the design of the memory systems and execution pipelines of both programmable GPUs (i.e., descendants of the fixed-function 3D rendering processors) and modern CPUs (i.e., descendants of the general-purpose Intel 80386 and related processors). However, since the micro-architectural ancestors of modern GPUs had very different programming models and workloads from those of modern CPUs, the two systems are able to make different assumptions about program behavior. Specifically, GPUs (both modern and original) assume a massively parallel workload with many fine-grained operations to be performed independently and with little programmatic specification of order among the individual operations. In graphics workloads, these fine-grained operations are the projective-geometric or texture-interpolation computations used to render 3-D scenes onto a 2-D screen. Contrarily, CPUs (modern and original) are designed primarily to execute sequentially-specified programs. Until circuit density reached power dissipation limits and forced multi-core designs as mentioned above, the only parallelism exploited by CPUs was the implicit parallelism among instructions within a single instruction stream. These different sets of assumptions have important consequences for the design choices made in these two classes of processors. The remainder of our discussion of processor architecture will describe processor design in general, but note the important differences in CPU and GPU systems.

The memory systems of modern processors are deeply hierarchical. Dynamic Random Access Memory (DRAM) is the highest level of the hierarchy that we consider in the design of high-performance programs. In general, it is highly desirable to avoid using magnetic

(hard disk) storage during program execution, either explicitly via file input/output or implicitly via page-faulting. DRAM storage typically resides on different discrete silicon parts from the processor cores, except in a few exotic architectures such as IBM's BlueGene [45]. Consequently, DRAM access latency is very high – typically several hundred processor cycles. DRAM access bandwidth is typically several tens of Gigabytes per second on CPU systems, and several hundreds of Gigabytes per second on GPU systems. The difference is primarily due to more aggressive memory-controller design and to the higher clock rates of the Graphics DRAM. Also due to these higher clock rates, the capacity of DRAM on GPUs is substantially lower than that of CPUs: 1-6 GB rather than 4-128 GB. The lower levels of memory hierarchies are the caches, of which current-generation systems have 3 levels. Intel CPU systems refer to these levels as L3, L2, and L1 caches. The L3 cache is shared by all cores within a socket, and the L2 and L1 caches are private to each core. Nvidia's GPU systems [63] have a small L2 cache that is shared by all cores in a socket, and L1 caches that are private to each core. Additionally, GPU register files are very large – approximately the same size as the L1 cache. The Cuda [62] programming model allows much of a program's working set to be held in the register file, while large persistent data objects reside in DRAM. An important difference in the design of CPU and GPU systems is that CPU caches are coherent, while GPU caches are not. *Cache coherence* refers to the ability of caches private to different cores to automatically communicate data written by one core to the other. In a coherent cache system, threads can communicate implicitly by writing data to memory. When another thread reads the same memory location, the cache system will automatically transfer the data to the reading thread's cache. An incoherent cache system will not make such transfers, and the programmer is responsible for coordinating communication by other means.

The execution pipelines of CPUs and GPUs leverage similar micro-architectural techniques to provide parallelism. Both types of processors can integrate multiple sockets with preferentially accessible DRAM nodes (NUMA), and multiple processor cores are integrated into the sockets of both types of systems. CPU systems integrate from 2 to 8 processor cores per socket, while high-performance GPUs integrate as many as 30. Within each core of both CPUs and GPUs, multiple simultaneous threads of execution are able to share execution resources. Intel's implementation of multi-threading is called Hyperthreading, whereas Nvidia's Cuda system somewhat confusingly calls these threads *Thread Blocks*. Each core additionally includes many execution units, which are accessible in parallel via Single-Instruction Multiple Data (SIMD): the core's instruction fetch logic issues a single instruction to all execution units simultaneously. SIMD is a very efficient mechanism to provide parallel execution, as it allows amortization of expensive and complex instruction fetch and decode logic over many execution units. However the implementation of SIMD in CPUs is rather inflexible, and it is difficult to speed up most codes with implementations like SSE or AVX. The GPU implementation of SIMD is substantially more flexible, as GPU architectures provide a multi-threading abstraction for programming SIMD hardware. Individual *Cuda threads*, up to 1024 of which comprise a *Thread Block*, can execute arbitrarily different programs and

access arbitrary memory locations. However, the SIMD hardware is heavily optimized for the case that groups of 32 threads are executing the same instruction and accessing contiguous blocks of memory. Nvidia refers to this architecture as Single-Instruction-Multiple Thread (SIMT).

Chapter 3

Software Design for High-Performance MRI Reconstruction

3.1 Introduction

The inverse problem formulations of MRI reconstruction problems provide a convenient mathematical framework in which to understand and formulate iterative reconstruction algorithms. However, these formulations alone provide little guidance for the productive development of efficient implementations for modern parallel processors. This chapter discusses several aspects of the design of the software systems that implement these algorithms, using the language of Design Patterns [2, 34, 58, 47]. Design Patterns provide a medium by which to relate software design problems that arise in a variety of contexts. For example, the patterns we discuss in this chapter originated in the context of massively parallel scientific computing. Software implementation is as much art as science, and concrete comparisons of software design strategies are difficult. Patterns are a medium by which to reason about software design problems and to document and discuss their solutions. The space of potential MRI reconstruction algorithms spanned by the inverse problem formalism is very broad. It is difficult to claim that any particular software framework can support every possible reconstruction approach. However, the use of design patterns lends substantial generality to our approach. The parallelization and implementation strategy described in this chapter applies to algorithms throughout signal and image processing, and provides a clear approach to extending our current implementation described in the following chapters.

The discussion in this chapter primarily applies to recently proposed iterative MRI reconstructions, for example iterative SENSE [53], SPIRiT [57], and simultaneous sensitivity-image estimation [79]. Traditionally, MRI reconstructions have used only *direct* algorithms, which prescribe a finite sequence of operations for an exact (within machine precision) computation. The Fast Fourier Transform (FFT) is a direct algorithm, in that it performs a finite-length sequence of operations to compute its result. *Iterative* algorithms specify a

finite-length sequence of operations to improve an approximate solution: one cannot say *a priori* how many iterations are necessary for the procedure to converge to an exact solution. Typically the estimate-update procedure is described in terms of direct algorithms such as matrix multiplication or an FFT. Each iteration is thus at least as expensive as a direct reconstruction. There are two types of software design issues that we address in this work. Because the direct computations performed in each step of an iterative algorithm dominate runtime, we are concerned with their efficient parallel implementation. Because iterative algorithms are difficult to implement, debug, and test, we are concerned with their productive development. In particular, we must allow the computationally-demanding operations to be implemented and optimized in isolation from the numerically-sensitive iterative algorithms to enable the use of high-performance libraries.

The primary contribution of this chapter is a software architecture composed of two design patterns in order to enable the most computationally intense operations to be performed by highly optimized library routines. We call this strategy the Supermarionation software architecture, and we describe it in Section 3.3. In Section 3.5, we describe the Geometric Decomposition [58] pattern. Geometric Decomposition provides a strategy to parallelize the computations, while the Puppeteer pattern [70], which we discuss in Section 3.4, provides a strategy to decouple the iterative and direct algorithms. Our approach is similar to the design of the Portable, Extensible Toolkit for Scientific computing (PETSc) [4]. Many of the performance problems addressed by PETSc are specific to large-scale distributed memory systems. PETSc defines several fundamental objects that recur in scientific computing applications, for example vectors and matrices. PETSc then provides libraries of highly optimized computations performed with these objects, such as vector scatter/gather, matrix products, Jacobian computations, and linear solvers.

3.2 Design Patterns and Software Architecture

Software Design Patterns are general, re-usable solutions to common software design problems. In many cases it is impossible to describe these design issues in a formal mathematical framework. Even in cases where formal descriptions are possible, their rigidity limits their scope and applicability to real software systems. Patterns are natural-language descriptions of problems, and thus cannot be manipulated or reasoned about via formal mathematics. However, this same informality is the source of design patterns' generality, as patterns typically assume very little about the particular computations being performed. Software architectures are descriptions of the components of a software system, and the ways in which these components interact. Frequently the design of these components can be guided by design patterns. The interactions between the components can also be described by patterns, and these patterns are commonly called *architectural* patterns.

In the context of designing high-performance MRI reconstructions, we face two competing design problems whose solution we describe via design patterns. First, we wish to leverage

powerful iterative numerical algorithms to solve the inverse problem formulations of MRI reconstruction tasks. These algorithms are typically much more computationally intense than the direct, heuristic-based approaches that they seek to supplant. This computational intensity results from the repeated application of the operations that compose the inverse problem's observation function and regularizers. Since routine clinical imaging requires that reconstructions fit into the established radiology workflow, the software implementations of reconstruction algorithms must be very efficient and heavily optimized. Section 3.5 will discuss how the Geometric Decomposition [58] pattern can guide much of these lower-level performance-relevant design decisions. Second, the flexibility of NMR has produced myriad applications of these iterative reconstruction techniques, such as 4D-Flow reconstruction [42], Diffusion-weighted imaging [10], and functional MRI via the BOLD signal [76]. MRI is a very fertile ground for research and development of new diagnostic techniques: new applications and reconstruction algorithms arise frequently. It is infeasible to expect that reconstruction software can be redeveloped for each new application individually. The researchers developing the applications typically do not have software engineering expertise or software development experience, and ensuring correctness in the implementation of numerical algorithms is notoriously difficult. Thus the software used to implement reconstructions must be flexible and modular to enable a high degree of software re-use. We discuss how the Puppeteer design pattern guides the implementation of algorithms in a way to satisfy both the desire for software re-use and the desire to optimize computationally intense operations.

These design problems are not specific to MRI reconstruction. Stated in these general terms, nearly any computing application can fit the description above. Scientific computing has faced these problems for decades, and has produced a large body of literature describing solutions for performance and software design problems. In recent years, researchers in a number of application areas have produced cutting-edge results with algorithms that bear both theoretical and practical similarity to those used in iterative MRI reconstructions. The goal of this section is to provide a detailed description of the MRI reconstruction software design problems.

3.3 The Supermarionation Software Architecture

In this section, we describe our strategy for the parallel implementation of iterative MRI reconstructions. We call this software architecture "Supermarionation."[1] The Supermarionation architecture applies in any context where a reconstruction problem is to be formulated as an inverse problem and solved via an iterative algorithm. As mentioned above, this class of applications is very broad and extends to a variety of signal processing and scientific computing tasks. For example, applications as diverse as image de-blurring and de-noising [27]

[1]The term Supermarionation describes the style of puppeteering used originally in the 1960's British television show "Thunderbirds," and more recently in the satirical film "Team America: World Police Force." A common theme is a "race against time," hence re-use of the term in our context is appropriate.

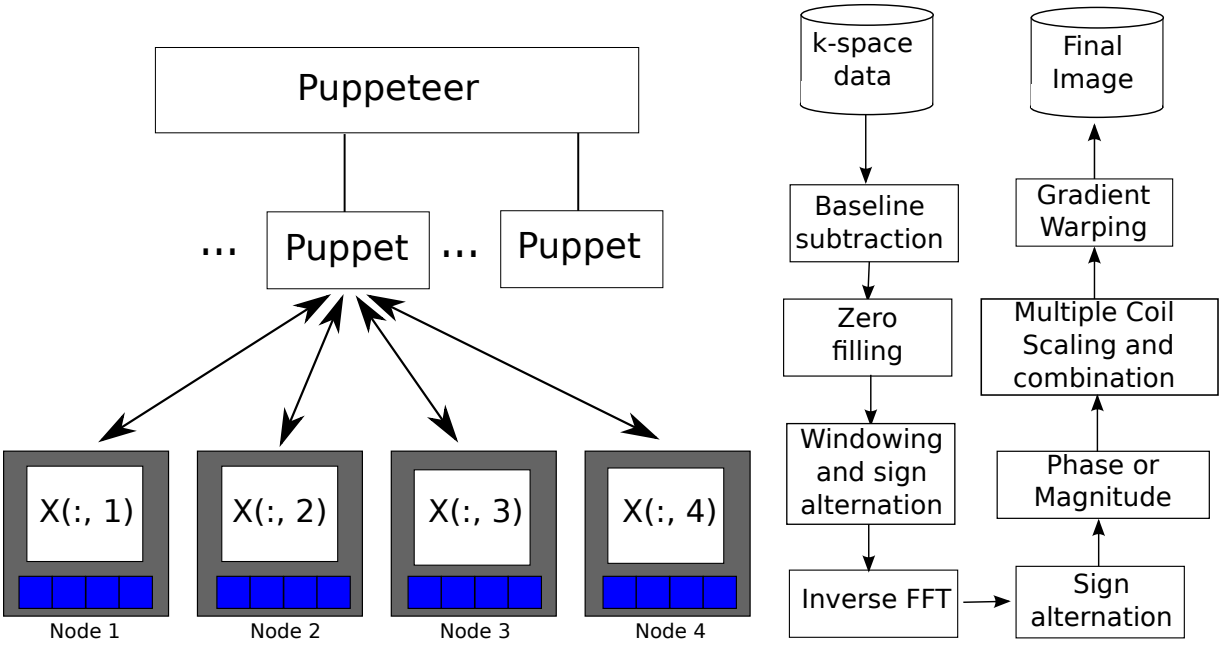

Figure 3.1: (left) The Supermarionation Architecture, which combines the Geometric Decomposition strategy for parallelizing computationally-intense operations with the Puppeteer strategy for implementing iterative algorithms. Each Puppet is parallelized according to the Geometric Decomposition of the data with which it interacts. The Puppeteer is responsible for ensuring the consistency of the Decompositions used for each puppet. (right) Example of an end-to-end direct Fourier reconstruction flowchart, reproduced partially from Bernstein *et al.* [10], Figure 13.8.

and face recognition [89] use similar techniques. For concreteness, our description is tailored towards the MRI reconstruction context.

Supermarionation is a composition of the two design patterns described in the following sections of this chapter. Section 3.5 describes the partitioning of data that drives the parallelization of computationally intense operations, while Section 3.4 describes a strategy for separating the parallelization of these operations from the implementation of the iterative algorithms that use them. The primary function of these two patterns is to describe the interactions between the important components of the software system that implements the reconstruction. The most computationally intense components are typically implemented as library routines, and the optimization of these routines are responsible for much of the high performance achievable in MRI reconstructions. However, data layout decisions as described by the Geometric Decomposition pattern have implications for all components in the system. The particular data layout and partitioning strategy chosen is a crucial aspect of component interactions. The modularization of sub-computations as described by the Puppeteer pattern limits the number of interactions which must be considered, and makes clear the role that library routines play in the overall system.

The Supermarionation architecture has several nice implications for the workflow of developing, debugging, and testing a reconstruction approach. The decoupling of the iterative algorithm from the underlying matrix and vector computations allows for more productive software development. The iterative algorithm can be developed and debugged entirely in isolation from its MRI application. Debugging and testing can be performed on small, synthetic problems for which a solution is analytically verifiable. Many theoretically provable properties of the algorithms, such as convergence rates and numerical stability, can be verified in this way. These properties are central to debugging and testing strategies for iterative algorithms. Similarly, each of the algorithm's subcomputations can be implemented, tested, and optimized independently. It may be desirable to implement multiple parallelizations, corresponding to alternative data layouts and partitioning strategies, in order to optimize for different data sizes or application contexts. Development of these implementations can be performed in isolation from each other as well as from the iterative algorithms. Moreover, the same optimized implementations can be re-used as Puppets in multiple iterative algorithms. Conversely, a single Puppeteer-driven implementation of an iterative algorithm can potentially be re-used with multiple data partitioning strategies. Using the Composite-Puppets implementation strategy as described in Section 3.6.6, the development of several operators can be performed in isolation even when they are to be composed into a single Puppet operation.

Typically, the end-to-end reconstruction software system will include several computational steps other than the solution of the reconstruction inverse problem. These steps may estimate additional system parameters via separate inverse problem formulations, or they may perform other image manipulations that solve particular medical imaging problems such as registration, gradient warping, or receiver bias correction [10]. Although some of these computations may in theory be incorporated into the inverse problem formulations, for prac-

tical reasons they are usually performed separately. Many useful image processing steps do not apply to all scans, and thus the desire for generality in a reconstruction system motivates the separation of the inverse-problem solver from other algorithms. As a result, the architecture of a reconstruction system to be deployed in a clinical setting is better described in a flowchart, also referred to as a Pipe and Filter architecture, such as the right-hand side of Figure 3.1.

For example, partial k-space acquisition [10] is a technique to accelerate scanning by sampling k-space asymmetrically. In the limiting case, samples are only acquired in one half-space. Homodyne reconstruction [10] is based on the assumption that the image of magnetization is real-valued. In that case, the Fourier conjugate-symmetry property of real signals can be used to estimate the missing half-space. In practice, MR images are not real, but exhibit slowly varying phase. Therefore, the conjugate-symmetry assumption is only approximately valid, and some samples of the other half-space must also be acquired to estimate this phase. The inverse problem formalism for reconstruction is general enough to include arbitrary k-space sampling patterns, including those with partially-missing half-spaces. One could regularize the solution with an approximate conjugate-symmetry assumption, and perform a simultaneous Homodyne/Compressive Sensing reconstruction. However, if one only has access to separate implementations of Homodyne and Compressive Sensing algorithms, then the reconstruction must perform them separately.

3.4 Puppeteer

As mentioned in Section 3.3, the Puppeteer pattern describes a strategy for decoupling the implementation of the sub-computations in an iterative reconstruction. The Puppeteer pattern [70] was originally used in multiphysics simulations to describe a strategy for decoupling sub-modules of a time-stepping numerical differential equation solver. The original work describes a climate simulation consisting of separate models for atmosphere, ocean, and land. The climate simulation implements a numerical solver for a differential equation defined in terms of the states of the individual models. Additionally, each model's state is updated via a separate numerical solution algorithm. The Puppeteer pattern describes the implementation of the interfaces between the climate-level simulator and the sub-models to avoid defining interfaces between the individual modules. Implementation and data-layout of the submodules are hidden behind the unified submodule interface. The term *Puppeteer* describes the implementation of the climate-level simulator. The term *Puppet* describes the submodules implementing the atmosphere, ocean, and land simulations. The role of the *Puppeteer* is to manipulate and control the *Puppets* and orchestrate any data transfers among them. This pattern is similar to the Mediator [34] pattern which has been described in more general object-oriented contexts. This approach is also very similar to that taken by the Portable, Extensible Toolkit for Scientific Computation (PETSc) [4]. PETSc also has re-usable, parallelized implementations of matrix, vector, and Jacobian operations.

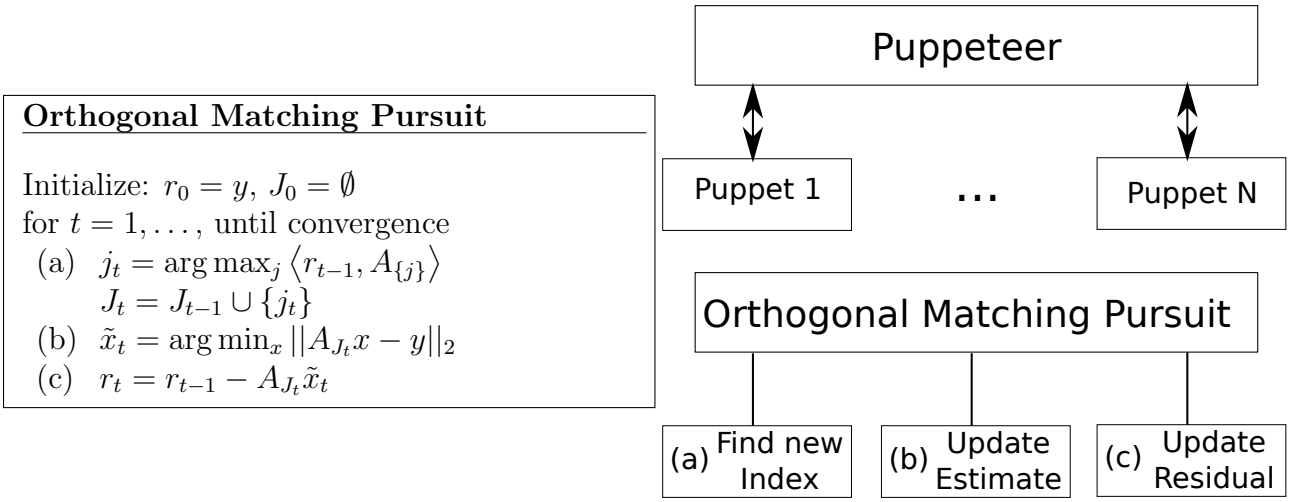

Figure 3.2: (top-right) General Puppeteer pattern. (left) The Orthogonal Matching Pursuit (OMP) algorithm for finding a sparse approximate solution x to the underdetermined linear system $Ax = y$. The set J contains indices of the columns of the measurement matrix A for which the reconstructed solution $\tilde{x}$ has non-zero components, and we denote by A_J the sub-matrix containing only the columns indicated by the set J. (bottom-right) The OMP algorithm represented as a Puppeteer. The three steps of the algorithm each are implemented independently, and the implementation of the OMP Puppeteer coordinates communication between them.

In our MRI reconstruction context, the Supermarionation architecture prescribes that the iterative inverse-problem solver be implemented in a similar fashion. The sub-computations of these solvers form the sequence of operations that update a solution estimate. Much as in the time-stepping differential equation solvers described by the original Puppeteer pattern, this sequence of operations will be computed repeatedly. The data objects accessed and modified by the sub-computations will be partitioned among the available processor cores according to one of the strategies described in Section 3.5.

For example, consider the implementation via the Puppeteer pattern of Conjugate Gradients on the Normal Equations (CGNE). CGNE solves a linear system $Ax = b$ in a least-squares sense, and requires computation of several vector-vector operations (dot products and scaled-additions) as well as the matrix-vector product Az (where z is a vector the same size as x) and the conjugate-transpose product A^*y (where y is a vector the same size as b). These computations should be encapsulated in Puppet objects in order to hide implementation details such as the strategy used to partition the vectors. Figure 3.2 describes another example, the Orthogonal Matching Pursuit (OMP) algorithm. Like CG, OMP also finds a solution to a linear system $Ax = b$. However, OMP finds a *sparse* solution, i.e. a solution with a small number of non-zeroes. The OMP algorithm iteratively performs three computations that update a solution estimate $\tilde{x}_t$ and a residual vector $r_t = A\tilde{x}_t - b$. First, OMP identifies the column of the measurement matrix $A_{\{j\}}$ with the highest correlation to the residual. Next, OMP computes a value for the j^{th} position in the solution estimate while simultaneously updating the values for all previously identified nonzeros in $\tilde{x}$. OMP achieves this by solving $A_{J_t}x - b$ in a least-squares sense, where A_{J_t} is the matrix induced by selecting only the columns of A for which $\tilde{x}$ has nonzeroes. Finally, OMP updates the residual to reflect the updated solution estimate. If OMP is implemented via the Puppeteer pattern, then these three subcomputations play the role of "Puppet". The implementation of the iterative OMP algorithm can be made agnostic of data layout, partitioning, and parallelization. However, the subcomputations operate directly on the data and must explicitly account for its layout.

In an MRI reconstruction context, the linear operator A will be a composition of the operations described in Section 3.6, such as Fourier and Wavelet transforms. When A is composed only of such operations, it requires no storage. However, in the general case the data structure representing A (or some sub-operation composed into A) must be partitioned in a way compatible with the decomposition of the data. If possible, the Puppeteer must be neither responsible for, nor aware of any distributed-memory communication required in computing matrix-vector products and vector-vector operations. Otherwise, the Puppeteer implementation cannot be re-used across different data decomposition strategies. However, when incompatible decompositions are used in two Puppets that use the same data object, it may be necessary for the Puppet to orchestrate the appropriate re-partioning.

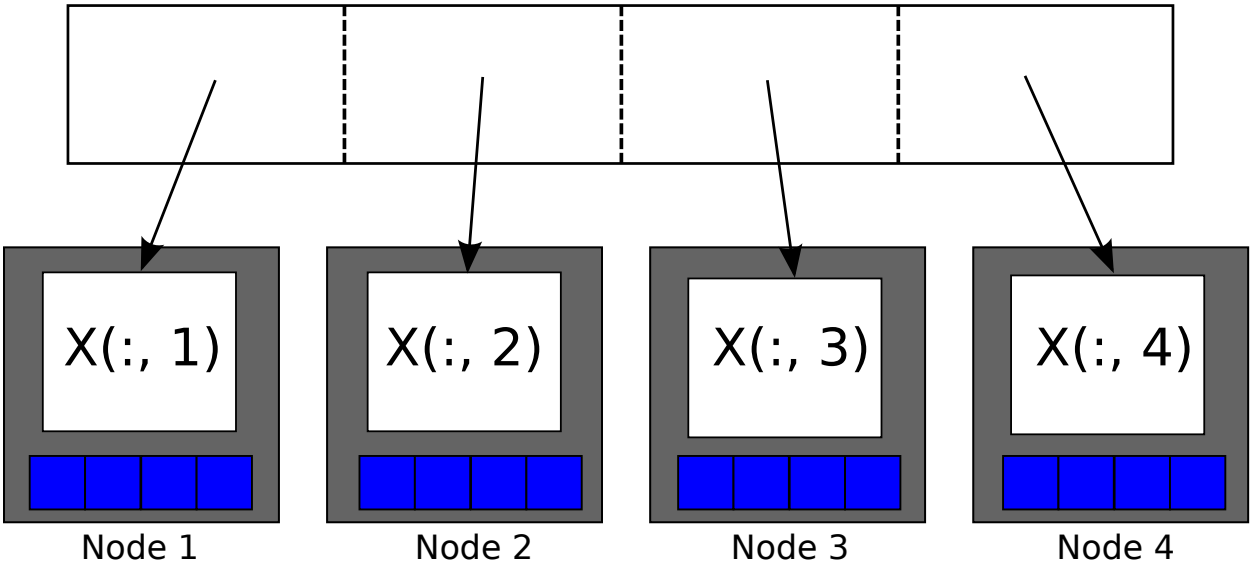

Figure 3.3: Illustration of the Geometric Decomposition pattern. Here, a 2-dimensional array X is decomposed across a 4-node distributed memory machine with multi-core shared-memory processors (depicted as blue boxes). The second dimension of X's index space is partitioned across the four nodes, so that Node i's memory contains the i^{th} column of X.

3.5 Geometric Decomposition

The Geometric Decomposition pattern describes parallelization of an algorithm via a partitioning of the underlying data on which the algorithm operates. In its original description [58], Geometric Decomposition primarily discussed distributed-memory parallelizations of nearest-neighbor operations. The original work describes these operations as *mesh computations*, and other scientific computing literature refers to these as *stencil* operations. These computations typically arise in the scientific computing literature when computing finite-differences in the iterative solution of differential equations. The relationship to MRI reconstruction is made clear by interpreting the finite-difference operations as convolutions, which arise frequently in signal-processing contexts like MRI. This same class of scientific computing shares many other algorithms with MRI, for example Fourier transforms and methods for solving linear systems.

The Geometric Decomposition pattern assumes the existence of a central object, set of objects, or data structure on which the algorithm performs its most performance-critical operations. In the finite-difference examples in the original pattern, this data structure is the array over which the finite differences are to be computed. Multiple related operations over these data structures contribute most to the computational complexity of the algorithm, and tend to occupy most of its runtime. These data structures tend to be the largest in the algorithm, as operations on small data structures rarely require substantial runtime. Per

Amdahl's law, performance-oriented software design should focus on these expensive operations first. While the algorithm may perform computations that do not involve the central data structure, their optimization should be considered only after ensuring satisfactory performance for the most expensive operations.

The Geometric Decomposition pattern prescribes that the implementation partition these data structures among the available processing threads. Each thread is said to "own" a partition, and is responsible for all computations that update (or produce) the values in that partition. In most cases the data structure to be partitioned is an array accessed via a multi-dimensional index: $\mathbf{i} = \{i_1, i_2, \ldots, i_D\} \in \{1, \ldots, N_1\} \times \ldots \times \{1, \ldots, N_D\}$. The Geometric partitioning strategy is to divide one or more of the D index spaces. As discussed in the following subsections, application context determines the choice of index space divisions that are most convenient and provide the highest performance. This simple strategy has also been described as an *output-driven* [36] parallelization, and typically simplifies synchronization: when all threads are modifying disjoint memory locations, they do not need to use mutual-exclusion primitives to ensure correctness.

In distributed-memory implementations, data resides in the same memory node as its owning thread. Consequently, Geometric Decomposition incurs communication cost when the update of one partition requires data from other partitions. Performance crucially depends on the ability to perform this communication efficiently. Inter-node communication is usually very expensive, as both the fixed per-message overhead and the per-byte transfer times are high. Partitions should thus be chosen to minimize communication. In shared-memory parallelizations, communication is manifest implicitly as cache and memory-system traffic. Additionally, the data partitioning is entirely logical, and may not require any data structure reorganization. Shared-memory communication typically requires substantially less programmatic overhead than does distributed-memory communication, which must be performed via explicit library calls. Distributed-memory communication also requires data to be collected into contiguous buffers before transfer over the message-passing interconnect. Thus there are substantial performance and software-productivity reasons to prefer shared-memory communication over distributed-memory communication. The systems on which MRI reconstructions are to be performed include both distributed- and shared-memory parallelism, and implementation choices can in some cases avoid the more expensive message-passing communication altogether.

We now discuss several ways in which the Geometric Decomposition pattern apply to MRI reconstructions. The differences in the four cases discussed below arise from the different scales of the classes of problems, and the resulting differences in the memory-system behavior of reconstruction algorithms.

3.5.1 2D and Real-Time Reconstruction

Reconstruction of 2D static images typically does not present substantial performance optimization problems. In most published reconstruction approaches, even prototype implemen-

tations developed in inefficient, high-level languages (e.g. Matlab) require only a few seconds of runtime for 2D static images. However, in real-time 2D imaging it is highly desirable that images be available during scanning. This is especially true for interventional applications. In these cases, the reconstruction algorithm must complete in the same time-frame as the real-time data acquisition. With highly efficient non-Cartesian or echo-planar trajectories, frame rates of 10–50 frames per second are possible [80]. If sophisticated iterative reconstruction algorithms are to be applied in this case, then achieving real-time performance can present a substantial implementation challenge. Nearly any proposed reconstruction approach can be adapted for either 2D or 3D reconstructions, and much of the same performance optimization discussion applies to both cases. The primary difference between the 2D case and the 3D case is the magnitude of the data involved, and thus the data partitioning strategies that are likely to improve performance will differ.

The amount of data involved in 2D reconstructions is substantially smaller than that for 3D reconstructions. For example, 512^2 is typically an upper bound to the image matrix size at which the data is to be reconstructed. The same Parallel Imaging coils are used in both 2D and 3D imaging, so the number of channels is 4–32 in both cases. With an image matrix of 512^2 each channel requires 2 MB of storage if computations are performed in single-precision, as is common in high-performance reconstruction studies. A 32-channel reconstruction would then require 64 MB per image matrix for which the algorithm must allocate storage. It is common for the highest-level caches on modern multicore CPUs to be several megabytes at least, and cache sizes continue to grow with Moore's law in the multi-core era. Approximately half of the die-area of high-performance microprocessors is devoted to these last-level caches. Current generation Intel Nehalem-EX processors include up to 24 MB of on-chip cache per socket, and shared-memory NUMA systems with up to 4 sockets are available. Consequently it is feasible that appropriate data partitioning for a shared-memory Geometric Decomposition parallelization to result almost entirely in in-cache computation. Care must be taken when partitioning the unit-stride (most rapidly-varying) dimension, as false-sharing of cache lines can substantially increase memory traffic and degrade performance.

However, the relatively small working set sizes limit the scalability of 2D reconstruction tasks. In particular, a distributed-memory partitioning of a 2D reconstruction is likely to suffer from very high synchronization and communication overheads. Such a partitioning may provide moderate performance gain, but a shared-memory parallelization allows inter-thread communication at much lower overheads. Thus the scale of a practical parallelization of a 2D reconstruction is limited by the size of shared-memory systems.

3.5.2 3D-Decoupled Reconstruction

Many k-space sampling trajectories can be designed to undersample in only one or two of the three spatial dimensions. The reconstruction can then leverage the third, fully-sampled dimension to extract additional parallelism and reduce overall computational complexity.

The simplest approach to designing Compressive Sensing pulse sequences is to modify a 3DFT Cartesian sequence to acquire a randomized subset of the (y, z) phase encodes. The sampling rate and receiver bandwidth in the readout direction x can be left unchanged.

Similarly, non-Cartesian sequences can be designed to provide irregular sampling in the (y, z) plane only. These sequences are commonly called stack-of-spiral and stack-of-stars [10], in contrast to fully 3-dimensional sampling trajectories that vary the sampling patterns across (y, z) planes. By preceding each non-Cartesian pulse sequence with a DFT-style phase encode in the x dimension, the same set of (k_y, k_z) sample locations can be acquired for an equally-spaced set of k_x values.

In these cases, the same set of (k_y, k_z) k-space locations are acquired for all values of k_x. The reconstruction can perform a Fourier transform along the fully-sampled dimension to decouple the (y, z) slices. This approach may not be SNR-optimal, as each individual reconstruction will have fewer samples over which to average noise. However it can have substantial performance benefits. All iterative reconstruction algorithms will be at least log-linear in the number of voxels in the reconstruction, due to the use of fast transform algorithms like the FFT or Wavelet transforms. Decreasing the size of a transform will provide a moderate decrease in computational complexity: the number of arithmetic operations performed will decrease by a factor of $\log N_x$, where N_x is the reconstruction image matrix size in the fully-sampled direction. Additionally this transformation reduces the size of the working set for a reconstruction, and the above discussion of efficient in-cache computation for 2D reconstruction applies. Moreover, decomposing the 3D reconstruction into independent 2D reconstructions for each of the (y, z) slices allows the 2D problems to be solved in parallel. Whereas the limited size of 2D reconstructions prevents efficient distributed-memory parallelization, the decoupled-3D reconstruction provides additional N_x-way parallelism that very naturally lends itself to distributed-memory systems. Geometrically Decomposing the 3D reconstruction by partitioning (y, z) slices among distributed-memory nodes is highly efficient, since no communication is required among the individual 2D reconstructions.

3.5.3 Full 3D Reconstruction

In the more general volumetric imaging case, a fully-sampled dimension does not exist. 3D sampling trajectories like cones [38] or kooshballs (radial) [10] do not permit decoupling along any dimension as discussed above. Consequently the reconstruction must implement a 3-dimensional parallelization of the operations performed by the iterative algorithm, such as Fourier and Wavelet transforms.

Fully 3-dimensional reconstructions can potentially have much higher memory footprints than the analogous readout-decoupled reconstruction described above. Containing this large in-memory working set may require the memory capacity of multiple nodes of a distributed-memory system. In a decoupled reconstruction, the number of 2D problems in flight can be chosen to limit working set size. In a full-3D reconstruction, data structures used in the algorithm must account for the entire data volume. For example, a 3D dataset may be

as large as 512^3 voxels in 32 separate parallel receiver channels. In single precision, each such multi-channel volume occupies 32 GB of DRAM. These arrays store the vectors defined by iterative algorithms, and most algorithms require storage for several vectors simultaneously. A typical distributed-memory cluster node may have 64 GB of memory capacity, and many nodes may be necessary to hold the algorithm's working set. Additionally, some parallel imaging reconstructions require $O(n_c^2)$ storage, where n_c is the number of parallel imaging channels. The quadratic factor stems from all-to-all k-space interpolations that GRAPPA [37]-like reconstructions perform. When these interpolations are performed in the image domain, the storage requirement may be as high as $O(n_c^2 \cdot n_v)$, where n_v is the number of voxels to be reconstructed. If implemented in k-space, the memory requirement is negligible but the computational complexity is substantially higher, since the interpolation must be performed as a convolution rather than a point-by-point multiplication. Parallel imaging coil arrays are growing in size over time, and techniques such as coil-compression [14] are crucial to the feasibility of full-3D parallel imaging reconstructions. This is especially true for GPU implementations, where memory capacity is substantially more limited.

There are two alternative data-partitioning strategies, and the optimal choice depends on the data size, which determines the amount of communication, as well as on the relative cost of communication on a particular reconstruction system. The Geometric Decomposition for a full-3D reconstruction can choose to partition the data along the voxel-wise indices or along the coil-wise indices. Since memory capacity demands that the data be partitioned among distributed-memory nodes, either index-partitioning scheme will produce message-passing communication. In cluster-scale parallelizations this message-passing will be performed over the node-to-node interconnect, while in GPU implementations it will be performed over the PCI-Express links between discrete graphics cards. These two types of inter-node links have similar performance characteristics, and our discussion applies equally to both cases.

Partitioning the data spatially (along the voxel-wise indices) will allow cross-coil image-domain interpolations to be performed without communication. When the voxels at a given location from all coils are resident on a single distributed-memory node, the cross-coil interpolation can compute weighted sums of them without accessing remote nodes. However, this partitioning strategy will require substantial communication during Fourier and Wavelet transforms, which are computed independently for each parallel imaging channel. Specifically, computing these transforms will require $O(n_c \cdot n_v)$ bytes to be communicated over the inter-node links. On a bus architecture like PCI-Express, this can be prohibitively expensive. On the other hand, partitioning the data along the coil dimension can potentially incur no communication during Fourier and Wavelet transforms, since a single 3D volume can easily fit in the several GB of memory at each node. Coil-partitioning will require $O(n_c^2 n_v)$ bytes of communication during all-to-all cross-coil interpolation, however. If the reconstruction uses reduced-rank implementations [51] of these cross-coil interpolations, communication is reduced to $O(n_c n_k n_v)$ bytes, where n_k is the rank of the cross-coil interpolator.

3.5.4 Multi-3D Reconstruction

The final case to discuss are reconstructions of multi-volume acquisitions. In these applications, several 3D multi-channel volumes have been acquired and some processing is to be done that potentially requires computations across all the volumes. There are at least two MRI applications that fit this description: 4D Flow and multi-contrast imaging. In 4D-flow acquisitions, data is acquired continuously with prospective gating. Data are acquired with motion-sensitizing gradients, and reconstruction attempts to reconstruct time-resolved dynamics within a volume. A reported application of 4D-flow to cardiac imaging [42] used prospective cardiac gating to control the points in time during the cardiac cycle at which data were acquired. In this application, the data were acquired using compressive sensing acceleration. A number of individual time-point volumes are produced by interpolating the acquired data. Reconstruction requires both estimation of non-acquired k-space samples from each time-point via a Parallel Imaging/ℓ_1-minimization reconstruction, but also estimation of flow information across the reconstructed volumes. Multi-contrast imaging [11] exploits the fact that in most clinical diagnostic applications, multiple images of the same object are acquired. Each image is weighted with different contrast mechanisms, for example before and after the injection of a contrast agent such as Gadolinium, or for multiple echo-times to produce different T2 weightings. Additionally each image is acquired with acceleration, for example via randomized phase-encode undersampling. Multi-contrast reconstructions assume that certain characteristics of the object do not change across imaging. For example, typically in brain imaging the patient's head is held immobile by the plastic cage containing the receive coils. In this case the anatomy does not shift substantially during the examination, and the locations of edges are consistent across the multiple images. Several proposed reconstruction approaches exploit this similarity across images to reconstruct each image with higher fidelity.

3.6 Algorithms

As discussed in Section 3.2, re-usable software is a crucial resource to the rapid development of new reconstructions. Arguably, the most difficult and time-consuming software to develop are the highly optimized library elements that perform the computationally expensive operations in MRI reconstructions. This section describes several of the most commonly used algorithms in MRI reconstructions, paying particular attention to the details of their implementation that pertain to the design patterns discussion above. All of the computations we describe here have applications in areas other than MRI reconstruction. For example, Fourier transforms and linear algebraic operations permeate all of numerical computing, and Wavelet transforms have many signal-processing applications – most notably in the JPEG 2000 standard for image compression [75].

3.6.1 Fourier Transforms

The Fast Fourier Transform is one of the most widely used and most well-known algorithms in numerical computing. It is frequently said to have had a larger impact on computational science than any other development. A number of freely available libraries provide high-performance implementations, and its efficient implementation has been well-studied in prior literature. Since the FFT is so well understood, our discussion of it will be brief. In the MRI reconstruction context, the Fourier transform is used to model data acquisition. In particular, one can show from the classical Bloch equation that the received NMR signal is equivalent to Fourier transform of the spatial distribution of transverse magnetization. Consequently, the simplest reconstruction methods rely on the ability to rapidly perform an inverse Fourier transform to recover the image. When the scan is not accelerated via undersampling, a single inverse Fourier transform is sufficient. In general, undersampled acquisitions require additional processing to remove aliasing, but all MRI reconstructions will require computation of a Fourier transform. For example, a Parallel Imaging reconstruction may estimate missing data directly in k-space. Subsequently, each channel's image is computed via a Fourier transform, and the multi-channel data combined with some other method (for example, sum-of-squares). If the Parallel Imaging reconstruction removes aliasing in the image-domain, then it must first compute the aliased images via the FFT.

Geometric Decomposition The FFT algorithm performs a series of "butterfly" operations, which perform a complex multiply-accumulate per sample. In the k^{th} step of the FFT, an butterfly is defined for a pair x_i and x_j as the updates $x'_i \leftarrow x_i + x_j \cdot \omega^k$, $x'_j \leftarrow x_i - x_j \cdot \omega^k$, where $\omega = e^{2\pi\imath/n}$ is the n^{th} root of unity. Every element in the n-vector is updated in each step. The stride $i - j$ depends on k, and in many formulations of the FFT $i - j = 2^k$. If the vector is partitioned among multiple processing threads, then the threads must synchronize between each butterfly step. However, in a multi-dimensional transform the FFT is performed along each dimension independently. For example, in a 2-dimensional transform the FFT is first performed for each row, and subsequently for each column. Since the FFT requires synchronization/communication between subsequent butterfly operations, it is usually more efficient to parallelize a multi-dimensional FFT over rows/columns, rather than within a single 1D FFT. Most available libraries include APIs for both parallelization strategies. In a distributed memory parallelization, a multi-dimensional FFT of Geometrically Decomposed data requires a data-exchange step. This data-exchange is is frequently called a "corner turn," and is equivalent to transposing a matrix whose rows have been distributed among memory nodes.

3.6.2 Gridding

Gridding refers to the application of the non-Uniform Fast Fourier Transform (nuFFT) algorithm in MRI reconstructions. Another chapter of this thesis describes the performance

optimization of the nuFFT in much greater detail, so our discussion here will also be brief. While MRI data are always acquired in the Fourier domain, the extreme flexibility of MRI allows for an arbitrary set of sample locations. In particular there is no restriction that the spacing of samples be uniform. Such samplings are called non-Cartesian, and enable highly efficient pulse-sequence design. Since the Cooley-Tukey decomposition requires an equispaced sampling, non-Cartesian acquisitions cannot be reconstructed via standard Fast Fourier transform libraries. The computation of the Fourier transform as a direct sum is unfeasibly expensive, and non-Cartesian reconstructions employ the nuFFT algorithm instead. The nuFFT computes a resampling of k-space onto the equispaced grid, enabling the subsequent use of an FFT. For computational feasibility, the interpolation kernel is truncated to a small window.

Geometric Decomposition Most of our discussion of data-driven parallelization has been agnostic of the particular index space which is to be partitioned. In most cases in MRI reconstruction, the data to be decomposed represents an equispaced sampling of a signal. For example, the multi-dimensional images produced in MRI reconstruction are frequently interpreted as equispaced sampling of $m_{xy}(\mathbf{r})$, the transverse magnetization over the field of view. In multichannel data, the final index is an integral index over receiver coils and the magnetization is written as $m_{xy}(\mathbf{r}, c) : \mathbb{R}^d \times \mathbb{Z} \to \mathbb{C}$. The integer indices $\mathbf{i} = \{i_1, \ldots, i_k\}$ correspond to the spatial positions directly: $\mathbf{r} = \delta \cdot \mathbf{i}$, where $\delta = \text{diag}(\{\delta_x, \delta_y, \delta_z\})$ is the spatial resolution of the scan. In the case of non-Cartesian samplings of k-space, the indices do not correspond directly to spatial locations. Typically, non-Cartesian k-space data are stored in readout-order. The least-significant index i_1 corresponds to the position in a readout, the middle indices $i_2, \ldots, i_{k-1}$ correspond to the readout in which a sample was taken, and the last index i_k corresponds to the channel index. Care must be taken in the Geometric Decomposition of the Gridding operation, as a partitioning of one of the index spaces (Cartesian vs. non-Cartesian) does not guarantee a disjoint partition of the other space during resampling. This fact necessitates synchronization or use of processor-specific instructions that guarantee atomicity of read-modify-write updates to the grid samples. Chapter 5 will also discuss performing the re-sampling as a matrix-vector multiplication, an implementation strategy that requires the storage of a large sparse matrix. The same sparse matrix is used across all parallel imaging channels, and must be replicated in all distributed-memory nodes if the k-space data is partitioned over channels.

Relation to N-Body Problems A recent performance study due to Obeid *et al.* [64] attempted to optimize the resampling step by noticing a superficial similarity to the N-Body problem, which arises in astrophysical and biological simulation. The N-Body problem is defined as the computation of interactions among a large number of point objects. In most applications the interaction is defined for all pairs of objects, and the number of objects is very large. The naïve approach to computing these interactions requires $O(n^2)$ time with

n point objects, although $O(n \log n)$ approximations exist. In some applications, one is willing to truncate the interaction and only compute interactions between pairs of a limited distance. Pre-sorting the point objects into spatially-contiguous "bins" allows one to identify nearby points, and avoid all-to-all distance computations. By analogy, Obeid *et al.* pre-sort the non-Cartesian sample locations into spatially-contiguous bins. The gridding resampling does not require computation of interactions among the non-equispaced sample locations, but rather between the non-equispaced locations and the equispaced grid locations. Obeid's motivation is to identify the set of grid locations each sample will affect in order to obviate synchronization in a parallel implementation.

3.6.3 Wavelet Transforms

The Wavelet transform is widely used in modern signal processing, and has a number of well known applications. Wavelets are frequently used as sparsifying bases in compressive sensing or de-noising applications, as Wavelet bases are well known to represent natural images compactly. Much like the Fourier transform, the Wavelet transform can be understood as a change of basis that can be computed rapidly via a fast algorithm. Even when the chosen Wavelet is not orthonormal, the log-linear algorithms can efficiently compute the Wavelet-domain coefficients of a signal. Orthogonality is necessary for the existence of an inverse transform. However, one can easily define the adjoint operator for non-orthogonal Wavelets. The same log-linear algorithm can be used to compute either the adjoint or the inverse. A single-level Wavelet decomposition separates the spectral content of the a 1-D signal x into two components containing the "details" (i.e. high-frequency information) and the "approximation" (i.e. low-frequency information). This separation is typically achieved via convolution with a quadrature-matched pair of high-pass and low-pass filters. Since each component contains half of the spectral information of the signal, they are sub-sampled by a factor of two after filtering. A 2D signal is decomposed into four components, containing the vertical/horizontal detail/approximation coefficients, and a general D-dimensional signal into 2^D components. Typically, a Wavelet transform performs multiple levels of decomposition.

Geometric Decomposition: The above discussion of Fourier transform parallelization applies similarly to the Wavelet transform. The fundamental operation in a Wavelet transform is not a butterfly operation, but rather the decimated convolution with the quadrature-pair of high-pass and low-pass filters. Different Wavelet bases differ only in the design of these filters. These filters are applied separably along each dimension of a multi-dimensional signal, providing a synchronization-free source for parallelism. However, synchronization is necessary between the multiple levels of Wavelet decomposition, just as synchronization was necessary between Butterfly steps of the Fourier transform.

3.6.4 Convolutions:

Convolutions arise in many MRI reconstruction tasks. We have seen one un-common case in the resampling step of the Gridding algorithm. This case is un-common in that the two signals to be convolved are sampled differently. In most cases the two signals are both sampled at equispaced locations, and the convolution is computed via a simple doubly-nested summation. Convolutions are very computationally intense when both signals have large support. The well-known Fourier-convolution identity, however, allows the implementation of convolutions as element-wise multiplication in the Fourier domain. In many MRI reconstruction applications, Fourier-implementation of convolutions is the highest performing option, as the Fourier transform of the signal may need to be computed for other reasons, for example to subsequently compute a Wavelet transform. In this case, use of the Fourier transform substantially reduces the computational complexity of the convolution, as the cost of the FFT is amortized over the Convolution and Wavelet transform.

Geometric Decomposition As discussed in Section 3.5 the original Geometric Decomposition pattern was primarily concerned with finite-difference operations, which are equivalent to convolutions with small-support kernels containing -1 for all values other than the central point. Thus the discussion of ghost-zone exchange in distributed memory parallelizations applies wholesale to the implementation of convolutions. However, ghost-zone exchange is unnecessary when the convolution is implemented via Fourier transforms.

3.6.5 Element-wise Operations

Although the other algorithms described in this section tend to consume most of the runtime of iterative MRI reconstructions, they are not the only operations for which parallelization are necessary. The geometric decomposition of the core data structures affects the implementation of all operations that access them, even ones that Amdahl's law might otherwise encourage us to ignore. Indeed, the operations that are asymptotically less expensive may become disproportionately expensive if an implementation fails to parallelize them. These operations include vector-vector operations, such as dot products, re-scalings, sums and differences – the well-known BLAS-1 operations. Additionally, many compressive sensing and de-noising algorithms call for thresholding operations which typically operate element-wise to set vector elements with small value to zero. Another important class of element-wise operations are the Data Parallel [41] computations, which includes the well-known Map-Reduce [23] parallel operations. In some contexts, external tools may aid in the parallelization of these operations. For example, the Copperhead compiler [17] is able to translate purely functional data parallel Python programs into highly efficient Cuda implementations.

3.6.6 Composite Operators

In most MRI applications, the inverse problem $F(x) = y$ that describes the reconstruction is defined in terms of an observation function $F(x)$ that is a composition of some of the operations above. For example, in Parallel Imaging reconstruction the system matrix performs an element-wise weighting of the data by the coil sensitivities, and them Fourier transforms each coil's data independently. The corresponding system matrix is then a composition of several element-wise operations and Fourier operators. The previous chapter of this thesis includes similar discussion for a variety of other reconstruction formulations, describing the system matrices for each case. The algorithms described in 3.6.7 are typically defined in terms of the inverse problems with composite system matrices. Hence the re-usability of the optimized implementations of Fourier, Wavelet, and Convolution operators in the Supermarionation architecture requires a small layer of "glue" to flexibly paste together the low-level operations into the system-matrices prescribed by the reconstruction formulations. The implementation of the compositing-layer may need to allocate additional buffers when the operator is a composition, rather than a "stacking" as in the Parallel Imaging case.

3.6.7 Iterative Algorithms

Here we discuss several iterative algorithms that can be implemented according to the Puppeteer pattern and used to solve certain inverse-problem formulations of MRI reconstruction. We choose to describe the three algorithms that arise in the examples we'll discuss in Section 3.7.

Projection Over Convex Sets (POCS): The POCS algorithm is a very simple algorithm to describe and implement, although due to its simplicity and generality it has no theoretically analyzable convergence rate. The POCS algorithm is defined in terms of a number of convex sets $S_1, \ldots, S_k$, each with a defined projection operator $\Pi_1, \ldots, \Pi_k$, and with a non-empty intersection $\cap_i S_i$ in which it is assumed the solution lies. That is, given an approximate solution x to the inverse problem, the projection $\Pi_i(x)$ is the element $y \in S_i$ with minimum distance $||y - x||$. In the i^{th} iteration, the POCS algorithm updates a solution estimate by projecting onto each of the sets: $x^{(i+1)} = (\Pi_1 \circ \cdots \circ \Pi_k)(x^{(i)})$. Any fixed point of this iteration is guaranteed to reside in the intersection of the sets. The implementation of POCS as a Puppeteer is simple: each projection operator must be defined as puppet, and the interface must simply support the application to an approximate solution x. Some projections can be implemented in a purely functional side-effect-free manner, in which case the puppeteer need not apply them in any particular order. However, in general they cannot be applied in parallel: all parallelization must be performed within the projection operators. In some cases, it may be desirable to re-use storage to minimize memory footprint. In this case, care must be taken to avoid conflicts when multiple projection operators use the same buffers.

Conjugate Gradients (CG): The well-known conjugate gradients algorithm has many variations, for example Conjugate-Gradients on Normal Equations (CGNE) or the Least-Squares QR (LSQR) algorithm. These algorithms rely on the Krylov-subspace of the system matrix to ensure conjugacy of a series of search directions $\{z_i\}$. In the i^{th} iteration, the solution estimate $x^{(i)}$ in the direction of z_i: $x^{(i+1)} = x^{(i)} + \alpha z_i$. CG can be applied either to a linear system $Ax = y$, in which case there is an analytical expression for α, or to non-linear systems, in which case CG is used to minimize a quadratic objective $||Ax - y||_2$ that approximates the behavior of the non-linear function. The implementation of CG as a Puppeteer bears much similarity to the implementation of many Krylov-subspace solvers, and also to the implementation of the PETSc library as described above. CG performs a number of vector-vector (BLAS-1) operations in addition to the matrix-vector product Ax. Iterative linear solvers for non-symmetric systems also must perform the conjugate-transpose product A^*y. The implementations of the vector-vector and matrix-vector operations should be parallelized according to the appropriate Geometric Decomposition, and the interfaces defined accordingly. In object-oriented languages that allow operator overloading, particularly elegant implementations of CG can be achieved via overloading the multiplication, addition, and subtraction operators. In C++, these correspond to implementing the `operator*`, `operator+`, and `operator-` methods.

Iteratively-Reweighted Gauss-Newton (IRGN): IRGN is a general method for solving non-linear systems. Similar to the non-linear CG mentioned above, IRGN minimizes a quadratic approximation to an arbitrary nonlinear function. Given a non-linear system $F(x) = y$, IRGN solves a linearized equation $\frac{\partial F}{\partial x_n} dx_n + F(x_n) = y$ for the step-size dx_n, where $\frac{\partial F}{\partial x_n}$ denotes the Jacobian matrix of the nonlinear function F. The solution estimate is updated as $x_{n+1} = x_n + dx_n$, and the linearized system solved iteratively until the original non-linear system is solved within satisfactory tolerance. The discussion of IRGN's implementation as a Puppeteer is very similar to the discussion for CG, since the majority of IRGN's computational intensity lies within the method chosen for solving the linear equation. Indeed CG could be used to solve this system in a least-squares sense, and the discussion would be made identical.

3.7 Examples

In this section, we discuss how the Supermarionation architecture applies to several example MRI reconstruction formulations. These applications are drawn from the MRI reconstruction literature, and were chosen to be representative of the variety of iterative reconstruction approaches that have been proposed.

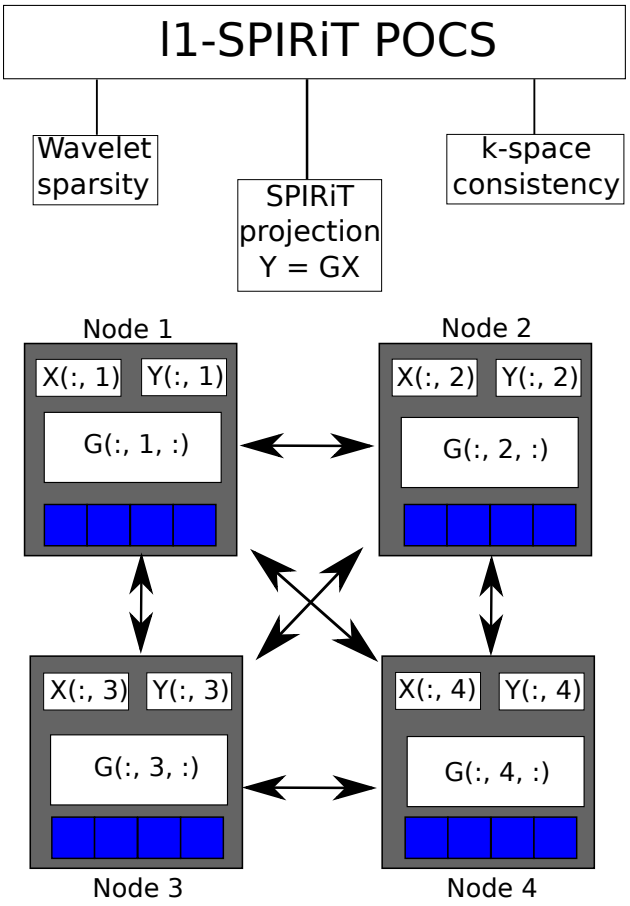

Figure 3.4: Supermarionation architecture of the Projections Onto Convex Sets (POCS) algorithm for solving the ℓ_1-SPIRiT reconstruction as described in Section 3.7.1. The three projection operators (Wavelet sparsity, k-space data consistency, and calibration-consistency) are shown as Puppets to the Puppeteer implementing the POCS algorithm. The Geometric Decomposition of the SPIRiT consistency projection, implemented as a Power Iteration, are shown. Individual channels of the image estimates X and Y and rows of the SPIRiT matrix G are distributed among four memory nodes.

3.7.1 POCS Algorithm for ℓ_1-SPIRiT

An in-depth performance analysis of our first example is described in much more detail in Chapter 4 of this thesis. However that discussion lacks an analysis of the software architecture used to design the implementation, opting instead to focus on the details of the performance optimization for current cutting-edge parallel hardware. Here, we focus on the description of ℓ_1-SPIRiT in terms of the Supermarionation software architecture.

Solving the ℓ_1 SPIRiT combined Parallel Imaging and Compressive Sensing reconstruction of Lustig *et al.* [57] requires finding an image x that simultaneously satisfies three constraints. First, x must be acquisition-consistent – i.e. it must be a solution to $\mathcal{F}x = y$, where $\mathcal{F}$ is a coil-by-coil subsampled Fourier transform operator, and y is the data acquired during the MRI scan. Second, x must be consistent with the SPIRiT Parallel Imaging calibration – i.e. it must solve $Gx = x$ for a matrix G that is computed prior to the POCS iterations. Finally, x must be Wavelet-sparse in an ℓ_1 sense as prescribed by Compressive Sensing theory.

To motivate the application of the POCS algorithm to this problem, we define three sets corresponding to the three constraints:

$$\begin{aligned} S_1 &= \{x : \mathcal{F}x = y\} \\ S_2 &= \{x : ||Gx - x||_2 < \varepsilon_2\} \\ S_3 &= \{x : ||Wx||_1 < \varepsilon_3\}. \end{aligned}$$

An exact projection onto S_1 is achieved by replacing the subset of Fourier coefficients of the estimate x corresponding to acquired samples with the values acquired during acquisition. A projection onto S_2 is defined by noting that $Gx = x$ is equivalent to requiring that x be a sum of eigenvectors of G with eigenvalues 1. Thus performing some iterations of an eigendecomposition algorithm such as the Power method [24] will serve as an approximate projection onto S_2. We can interpret the well-known soft-thresholding operation [27] used in de-noising and ℓ_1 minimization algorithms as a projection onto a set of wavelet-sparse images.

The Puppeteer implementation of the POCS algorithm must simply apply these three projections in some order. The geometric decomposition of the image data x drives the parallelization of all the operators that must be computed: Fourier transforms, multiplication by the G matrix, and Wavelet transforms. The less-expensive vector-vector operations that must be parallelized in the same manner are the soft-thresholding operation and the Fourier-space projection, which consists only of replacement of a subset of the Fourier coefficients with the acquired data. Computation of the G matrix is performed via a least-norm least-squares solver in a separate step. It can also be interpreted as the solution to an inverse problem, solved in a separate Pipe-and-Filter/flowchart stage.

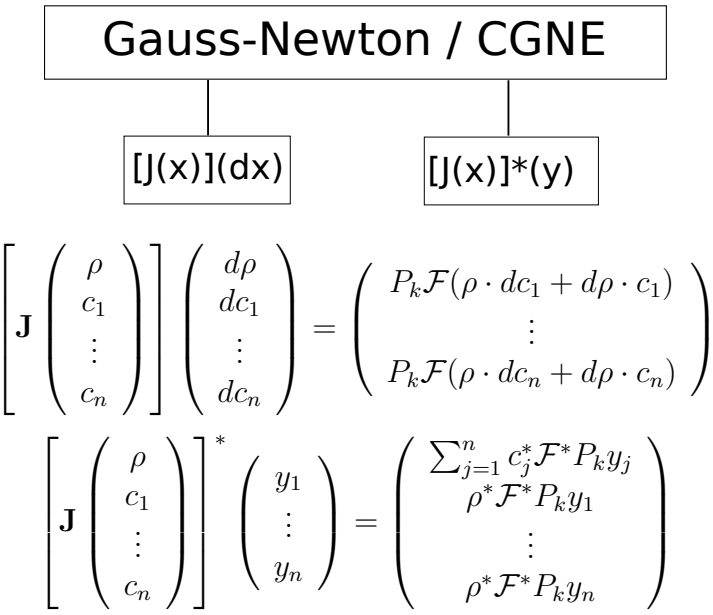

Figure 3.5: Supermarionation architecture of the Gauss-newton solver for the Parallel Imaging reconstruction described in Section 3.7.2. The forward and adjoint Jacobian operators are depicted as Puppets, and the formulae for their application shown. These operators are Composites of Fourier and element-wise operations, as discussed in Section 3.6.6

3.7.2 IRGN Algorithm for Non-Linear Parallel Imaging Reconstruction

The non-linear parallel imaging reconstruction of Uecker *et al.* [79] formulates the simultaneous estimation of the image and the coil sensitivity profiles as a non-linear inverse problem. The parameters to be estimated are $x = (\rho, c_1, \cdots, c_n)'$, where ρ is the transverse magnetization and x_i is the sensitivity profile of the i^{th} coil. The non-linear operator $F(x)$ computes the subsampled Fourier transform of the element-wise product of each c_i with ρ, and the inverse problem estimates x given the acquired samples y via inversion of $F(x) = y$. Uecker *et al.* solve this problem via the IRGN algorithm described above. Similarly to the ℓ_1-SPIRiT case, the Geometric Decomposition of the data x drives the parallelization of the operations that must be performed. Specifically, the algorithm must solve for the step-size dx via the linearized system. An iterative Krylov-subspace method can be used to solve this system, requiring application of the Jacobian operator and its adjoint. The Jacobian and its adjoint are compositions of element-wise and Fourier operations.

3.8 Conclusion

We have presented the Supermarionation software architecture to guide the parallel implementation of the inverse problems that arise in MRI reconstruction. This architecture provides a strategy for parallelizing the operations in a reconstruction algorithm based on a Geometric Decomposition of the underlying data, and a strategy for decoupling the implementation of the performance-optimized computationally-intense operators from the iterative algorithm chosen to solve the inverse problem. We described a set of algorithms that are used in both aspects of the architecture (performance-optimized operators and iterative algorithms), and several examples of reconstruction algorithms described as Supermarionation instances.

Chapter 4

ℓ_1-SPIRiT: Scalable Parallel Implementation and Clinically Feasible Runtime

4.1 Introduction

In Chapter 3, we discussed several software design issues inherent in the implementation of high-performance MRI reconstructions. One of the example reconstructions we mentioned was a Projections Over Convex Sets (POCS) algorithm for ℓ_1-SPIRIT, which combines Compressive Sensing and Parallel Imaging reconstruction approaches into a single iterative framework. We discussed the implementation of this reconstruction system using the Supermarionation software architecture, which composes a Puppeteer [70] strategy for implementing iterative algorithms with a Geometric Decomposition [58] strategy for parallelizing computationally intense operations. In this chapter, we focus on the lower-level implementation issues necessary for achieving clinically-feasible runtimes for ℓ_1-SPIRiT.

Imaging speed is a major limitation of MR Imaging, especially in comparison to competing imaging modalities such as Computed Tomography (CT). MR allows much more flexible contrast-generation and does not expose patients to ionizing radiation, and hence does not increase risk of cancer. However, other imaging modalities are substantially more popular, because MR scans are slow, expensive, and in some cases less robust. Patient motion during long scans frequently causes image artifacts, and for uncooperative patients, like children, anesthesia is a frequent solution. Acquisition time in MRI can be reduced by faster scanning or by subsampling. Parallel imaging [71, 67, 37] is a well-established acceleration technique based on the spatial sensitivity of array receivers. Compressed sensing (CS) [16, 26, 55] is an emerging acceleration technique that is based on the compressibility of medical images. Attempts to combine the two have mostly focussed on extensions of iterative SENSE [66] with SparseMRI [55]. In [12] Block *et al.*, added total-variation to a SENSE reconstruction

from radial sampling, Liang *et al.*, in [52] showed improved acceleration by first performing CS on aliased images and then applying SENSE to unfold the aliasing, Otazo *et al.* used compressed sensing with SENSE to accelerate first-pass cardiac perfusion [65]. More recently [78, 87] have presented some improvements, again, using an extension of SENSE. The difficulty in estimating exact sensitivity maps in SENSE has created the need for autocalibrating techniques. One class of autocalibrating algorithms extends the SENSE model to joint estimation of the images and the sensitivity maps [90, 79]. Combination of these approaches with compressed sensing have also been proposed. Knoll *et al.* [49] proposed a combination with Uecker's non-linear inversion and Huang *et al.* [43] proposed a self-feeding SENSE combined with compressed sensing.

A different, yet very popular class of autocalibrating techniques are methods like GRAPPA [37] that do not use the sensitivity maps explicitly. In [57] we proposed an optimized iterative method, SPIRiT, and demonstrated the combination with non-linear regularization. In [54] we presented and extension, ℓ_1-SPIRiT, that synergistically combines SPIRiT with compressed sensing and in [82, 81] we presented more details and clinical results in pediatric patients.

The combination of compressed sensing with parallel imaging has the advantage of improved image quality, but it comes at a cost. These algorithms involve substantially more computation than direct or iterative linear reconstructions.

In this chapter we discuss the ℓ1-SPIRiT reconstruction. ℓ1-SPIRiT solves a constrained non-linear optimization over the image matrix. The non-linearity of this optimization necessitates an iterative reconstruction, and we describe our simple and efficient POCS algorithm in Secion 4.3.

A recent trend in MRI has been to accelerate reconstructions by implementing and optimizing them for massively parallel processors. Silicon manufacturing technology has recently experienced the end of a trend that produced the incredible pace of comptuational speed during the 1990's [30]. In the past decade, all major microprocessor vendors have increased the computational throughput of their designs by introducing programmer-visible parallelism. Intel and AMD provide 4-16 CPU cores per socket, and GPGPUs typically have 16-32 massively multithreaded vector cores per socket. In each case, the computational throughput of the processor is proportional to the number of cores, and future designs will have larger numbers of cores.

This chapter discusses the massively parallel implementation of ℓ1-SPIRiT on these processors. The resulting sub-minute runtimes demonstrate that computational expense is not a substantial obstacle to clinical deployment of ℓ1-SPIRiT. Many previous works have demonstrated substantial improvement in reconstruction runtime using GPUs and multi-core CPUs as parallel execution platforms. Chang and Ji [18] demonstrated multi-channel acceleration by solving SparseMRI reconstruction separately for each channel and reporting 1.6-2.0 acceleration using 4 cores. More recently Kim *et al.* [48] present a high-performance implementation of a SENSE based compressive sensing reconstruction, describing many low-level optimizations that apply for both CPU and GPU architectures.

Stone *et al.* [74] describe the implementation of an iterative reconstruction using the Conjugate Gradient (CG) algorithm to solve regularized linear reconstructions for non-cartesian trajectories. Their implementation relies on a highly optimized GPU implementation of a non-uniform Fourier transform (NDFT) to perform sub-minute non-Cartesian reconstructions. Wu *et al.* [88, 93] have generalized this work to model other acquisition effects in the NDFT, such as off-resonance and sensitivity encoding. Several other works have discussed the GPU implementation of Gridding [7], a highly accurate NDFT approximation. Sørensen *et al.* [73] describe an algorithm for obviating potentially expensive synchronization in a GPGPU implementation of Gridding. Following their approach, Gregerson [36] discusses the performance trade-offs of different parallelization strategies for the gridding interpolation. Obeid *et al* [64] use a spatial-partitioning approach to optimize gridding interpolation, and reporting 1-30 second runtimes. Nam *et al.* [1] describe another gridding implementation achieving sub-second interpolations for highly undersampled data. Several other works have presented GPU implementations of Parallel Imaging (PI) reconstructions with clinically-feasible runtimes. Roujol *et al.* [69] describe GPU implementation of temporal sensitivity encoding (TSENSE) for 2D interventional imaging. Sorenson *et al.* [72] present a fast iterative SENSE implementation which performs 2D gridding on GPUs. Uecker [79] describes a GPU implementation of a non-linear approach to estimate PI coil sensitivity maps during image reconstruction.

This work presents the parallelization of an autocalibrating approach, ℓ1-SPIRiT, via multi-core CPUs and GPUs and the resulting clinically-feasible reconstruction runtimes. Moreover we discuss the approach taken to parallelizing the various operations within our reconstruction, and the performance trade-offs in different parallelization strategies. Additionally, we discuss the data-size dependence of performance-relevant implementation decisions. To our knowledge, no previous works have addressed this issue.

4.2 iTerative Self-Consistent Parallel Imaging Reconstruction (SPIRiT)

SPIRiT is a coil-by-coil autocalibrating parallel imaging method and is described in detail in [57]. SPIRiT is similar to the GRAPPA parallel imaging method in that it uses autocalibration lines to find linear weights to synthesize missing k-space. The SPIRiT model is based on self-consistency of the reconstructed data with the acquired k-space data and with the calibration.

SPIRiT is an iterative algorithm in which in each iteration non-acquired k-space values are estimated by performing a linear combination of nearby k-space values. The linear combination is performed using both acquired k-space samples as well as estimated values (from the previous iteration) for the non-acquired samples. If we denote x_i as the entire k-space grid of the i^{th} coil, then the consistency criterion has a form of a series of convolutions

with the so called SPIRiT kernels g_{ij}. The SPIRiT kernels are obtained by calibration from auto calibration lines similarly to GRAPPA. If N_c is the total number of channels, the calibration consistency criterion can be written as

$$x_i = \sum_{j=1}^{Nc} g_{ij} * x_j.$$

The SPIRiT calibration consistency for all channels can be simply written in matrix form as

$$x = Gx,$$

where x is a vector containing the concatenated multi-coil data and G is an aggregated operator that performs the appropriate convolutions with the g_{ij} kernels and the appropriate summations. As discussed in [57], the G operator can be implanted as a convolution in k-space or as multiplication in image space.

In addition to consistency with the calibration, the reconstruction must also be consistent with the acquired data y. This can be simply written as

$$y = Dx,$$

where D is an operator that select the acquired k-space out of the entire k-space grid. In [57] two methods were proposed to find the solution that satisfies the constraints. Here we would like to point out the projection over convex sets (POCS) approach which uses alternate projections that enforce the data consistency and calibration consistency. In this chapter we extend the POCS approach to include sparsity constraints for combination with compressed sensing.

As previously mentioned, the convolution kernels $g_{i,j}$ are obtained via a calibration from the densely sampled auto-calibration region in the center of k-space, commonly referred to as the Auto-Calibration Signal or ACS lines. In the reconstruction we would like to find x that satisfies $x = Gx$. However in the calibration x is known and G is unknown. We compute the calibration in the same way as GRAPPA [37], by fitting the kernels g to the consistency criterion $x = Gx$ in the ACS lines. Due to noise, data corruption, and ill-conditionedness, we solve this fit in an ℓ_2-regularized least-squares sense. We compute the calibration once, prior to image reconstruction. Calibration could potentially be improved via a joint estimation of the kernels and images, as has been presented by Zhao *et al.* as Iterative GRAPPA [92] and in SENSE-like models by Ying *et al.* [90] and Uecker *et al.* [79]. Joint estimation is more computationally expensive, but its runtime could be improved with techniques similar to those discussed in this thesis.

We can therefore solve for the g_{ij}'s by reformatting the data x appropriately and solving a series of least-squares problems to calibrate g_{ij}. This procedure is similar to calibrating GRAPPA kernels.

4.3 $\ell 1$-SPIRiT Reconstruction

Variations of the $\ell 1$-SPIRiT reconstruction have been mentioned in several conference proceedings [54, 60, 51, 82]. More detailed descriptions are given in [57] and in [81]. But for the sake of completeness and clarity we include here a detailed description of the variant that is used in this chapter.

$\ell 1$-SPIRiT is an approach for accelerated sampling and reconstruction that synergistically unifies compressive sensing with auto-calibrating Parallel imaging. The sampling is optimized to provide the incoherence that is required for compressed sensing yet compatible to parallel imaging. The reconstruction is an extension of the original SPIRiT algorithm that in addition to enforcing consistency constraints with the calibration and acquired data, enforces joint-sparsity of the coil images in the Wavelet domain. Let y be a the vector of acquired k-space measurements from all the coils, F a Fourier operator applied individually on each coil-data, D a subsampling operator that chooses only acquired k-space data out of the entire k-space grid, G an image-space SPIRiT operator that was obtained from auto-calibration lines, Ψ a wavelet transform that operates on each individual coil separately. ℓ_1-SPIRiT solves for the multi-coil images concatenated into the vector x which minimizes the following problem:

$$\begin{aligned} \text{minimize}_x \quad & \text{Joint}\ell_1(\mathbf{\Psi} x) && (4.1)\\ \text{subject to} \quad & \mathbf{DF}x = y && (4.2)\\ & \mathbf{G}x = x && (4.3) \end{aligned}$$

The function Joint$\ell_1(\cdot)$ is a joint ℓ_1-ℓ_2-norms convex functional and is described later in more detail. Minimizing the objective (4.1) enforces joint sparsity of wavelet coefficients between the coils. The constraint in (4.2), is a linear data-consistency constraint and in (4.3) is the SPIRiT parallel imaging consistency constraint. The Wavelet transform [13] $\mathbf{\Psi}$ is well-known to sparsify natural images, and thus used frequently in Compressive Sensing applications as a sparsifying basis. Just as the Fourier transform, it is a linear operation that can be computed via a fast $O(n \log n)$ algorithm.

As previously mentioned, in this work we solve the above problem via a an efficient POCS algorithm, shown in Figure 4.1. It converges to a fixed-point that satisfies the above constraints, often within 50-100 iterations. This algorithm does not solve the constrained minimization exactly, but instead minimizes the related Lagrangian objective function.

4.3.1 Joint-Sparsity of Multiple Coils

We perform soft-thresholding on the Wavelet coefficients to minimize the $\ell 1$-objective function (4.1). The soft-thresholding function $\mathcal{S}_\lambda(x)$ is defined element-wise for $x \in \mathbb{C}$ as:

$$\mathcal{S}_\lambda(x) = \frac{x}{|x|} \cdot \max(0, |x| - \lambda)$$

POCS Algorithm for ℓ_1-SPIRiT

x_k - image estimate after k$^{\text{th}}$ iteration
y - acquired data
F - multi-coil Fourier transform operator
G - SPIRiT operator
Ψ - multi-coil Wavelet transform operator
D - subsampling operator choosing acquired data
$\mathcal{S}_\lambda$ - Joint Soft-thresholding

$\mathbf{G} \leftarrow \text{AutoCalibrate}(y)$
Initialize $x_0 \leftarrow \mathbf{F^{-1}} D^T y$
for $k = 1, 2, \ldots$ until convergence:
(A) $m_k \leftarrow \mathbf{G} x_{k-1}$
(B) $w_k \leftarrow \mathbf{\Psi^{-1}} \mathcal{S}_\lambda \{\mathbf{\Psi} m_k\}$
(C) $x_k \leftarrow \mathbf{F}^{-1} \left[(I - D^T D) (\mathbf{F} w_k) + D^T y \right]$

Figure 4.1: The POCS algorithm. Line (A) performs SPIRiT k-space interpolation, implemented as voxel-wise matrix-vector multiplications in the image domain. Line (B) performs Wavelet Soft-thresholding, computationally dominated by the forward/inverse Wavelet transforms. Line (C) performs the k-space consistency projection, dominated by inverse/forward Fourier transforms.

where $|x|$ is the complex modulus of x. The parameter λ estimates the amplitude of noise and aliasing in the Wavelet basis, and the soft-thresholding operation is a well-understood component of many de-noising [27] and compressive sensing algorithms [22]. As pointed out in prior works [82], we approximate translation-invariance in our Wavelet transforms. Translation invariance reduces block-like artifacts [32], but a translation invariant Wavelet transform is very computationally expensive. The approximation we use is effective at removing the artifacts, but has a negligible computational overhead.

The individual coil images are sensitivity weighted images of the original image of the magnetization. We assume that coil sensitivities are smooth and do not produce spatial shift of one coil's image relative to another. Edges in these images appear in the same spatial

position, and therefore coefficients of sparse transforms, such as wavelets, exhibit similar sparsity patterns. To exploit this, we use a joint-sparsity model[82, 81]. In compressed sensing, sparsity is enforced by minimizing the ℓ_1-norm of a transformed image. The usual definition of the ℓ_1-norm is the sum of absolute values of all the transform coefficients, $\sum_c \sum_r |w_{cr}| = \sum_c \sum_r \sqrt{|w_{rc}|^2}$, where c is the coil index and r is the spatial index. In a

joint-sparsity model we would like to jointly penalize coefficients from different coils that are at the same spatial position. Therefore we define a joint ℓ_1 as:

$$\text{Joint}\ell_1(w) = \sum_r \sqrt{\sum_c |w_{rc}|^2}$$

In a joint ℓ_1-norm model, the existence of large coefficient in one of the coils, protects the coefficients in the rest of the coils from being suppressed by the non-linear reconstruction. In the POCS algorithm joint sparsity is enforces by soft-thresholding the magnitude of the wavelet coefficients across coils, at a particular position.

4.3.2 Computational Complexity

If n_c is the number of PI channels and v is the number of voxels per PI channel, the computational complexity of our algorithm is:

$$O\left(C_C \cdot n_c^3 + T \cdot ((C_W + C_F) \cdot n_c v \log v + C_S \cdot n_c^2 v)\right)$$

where T is the number of iterations the POCS algorithm performs. The algorithm often converges with sufficient accuracy within 50-100 iterations. The constants C_W, C_F, C_S, and C_C indicate that the relative computational cost of the Wavelet transforms, Fourier transforms, and SPIRiT interpolation and calibration are heavily dependent on input data size. Section 4.7 presents more detailed runtime data.

The n_c^3 term represents the SPIRiT calibration, which performs a least-norm, least-squares fit of the SPIRiT model to a densely sampled autocalibration region in the center of k-space. Solving each of these systems independently leads to an $O(n_c^4)$ algorithm, which is prohibitively expensive for large coil arrays. However, the n_c linear systems are very closely related and can be solved efficiently with only a single Cholesky decomposition of a square matrix of order $O(n_c)$, hence the $O(n_c^3)$ complexity of our method. See Section 4.5 for a complete derivation of the algorithm. This derivation can potentially be used to accelerate the computation of GRAPPA kernels as well.

The $n_c v \log v$ term represents the Fourier and Wavelet transforms, and the $n_c^2 v$ term represents the image-domain implementation of the k-space SPIRiT interpolation. This k-space convolution is implemented as multiplication in the image domain, hence the linearity in v of this term. Due to the $O(n_c^2 v)$ complexity, SPIRiT interpolation is asymptotically the bottleneck of the POCS algorithm. All other operations are linear in the number of PI channels, and at worst log-linear in the number of voxels per channel. There are several proposed approaches that potentially reduce the complexity of the SPIRiT interpolation without degrading image quality. For example, ESPIRiT [51] performs an Eigen decomposition of the $\mathbf{G}$ matrix, and uses a rank-one approximation during POCS iterations. Also, coil array Compression [15, 91] can reduce the number of parallel imaging channels to a small

constant number of *virtual* channels. Our software includes implementations of both of these approaches, and in practice the ℓ1-SPIRiT solver is rarely run with more than 8 channels.

One could solve the ℓ_1-SPIRiT reconstruction problem (Equations 4.1-4.3) via an algorithm other than our POCS approach, for example non-linear Conjugate Gradients (NLCG). The computational complexity of alternate algorithmic approaches would differ only in constant factors. The same set of computations would still dominate runtime, but a different number of iterations would be performed and potentially a different number of these operations would be computed per iteration. End-to-end reconstruction times would differ, but much of the performance analysis in this work applies equally well to alternate algorithmic approaches.

4.4 Fast Implementation

The POCS algorithm is efficient: in practice, it converges rapidly and performs a minimal number of operations per iteration. Still, a massively parallel and well-optimized implementation is necessary to achieve clinically feasible runtimes. A sequential C++ implementation runs in about ten minutes for the smallest reconstructions we discuss in this chapter, and in about three hours for the largest. For clinical use, images must be available immediately after the scan completes in order to inform the next scan to be prescribed. Moreover, time with a patient in a scanner is limited and expensive: reconstructions requiring more than a few minutes of runtime are infeasible for on-line use.

In this section, we discuss the aspects of our reconstruction implementation pertinent to computational performance. While previous works have demonstrated the suitability of parallel processing for accelerating MRI reconstructions, we provide a more didactic and generalizable description intended to guide the implementation of other reconstructions as well as to explain our implementation choices.

4.4.1 Parallel Processors

Many of the concerns regarding efficient parallel implementation of MRI reconstructions are applicable to both CPU and GPU architectures. These two classes of systems are programmed using different languages and tools, but have much in common. Figure 4.2 establishes a four-level hierarchy that one can use to discuss parallelization decisions. In general, synchronization is more expensive and aggregate data access bandwidth is less at "higher" levels of the hierarchy (i.e., towards the top of Figure 4.2). For example, Cuda GPGPUs can synchronize threads within a core via a `__syncthreads()` instruction at a cost of a few processor cycles, but synchronizing all threads within a GPU requires ending a grid launch at a cost of $\approx 5\mu$s, or 7,500 cycles. CPU systems provide less elaborate hardware-level support for synchronization of parallel programs, but synchronization costs are similar at corresponding levels of the processor hierarchy. On CPUs, programs synchronize via software barriers and

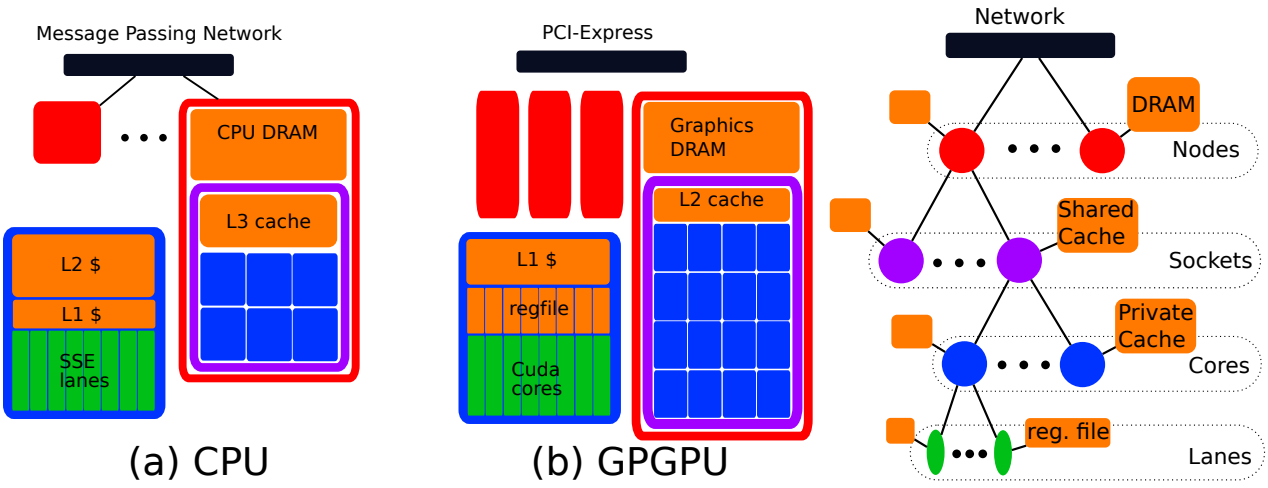

Figure 4.2: The four-level hierarchy of modern parallel systems. *Nodes* contain disjoint DRAM address spaces, and communicate over a message-passing network in the CPU case, or over a shared PCI-Express network in the GPU case. *Sockets* within a node (only one shown) share DRAM but have private caches – the L3 cache in CPU systems and the L2 cache in Fermi-class systems. Similarly *Cores* share access to the Socket-level cache, but have private caches (CPU L2, GPU L1/scratchpad). Vector-style parallelism within a core is leveraged via *Lanes* – SSE-style SIMD instructions on CPUs, or the SIMT-style execution of GPUs.

task queues implemented on top of lightweight memory-system support. With respect to data access, typical systems have $\approx$ 10 TB/s (10^{13} bytes/s) aggregate register-file bandwidth but only $\approx$ 100 GB/s (10^{11} bytes/s) aggregate DRAM bandwidth. Exploiting locality and data re-use is crucial to performance.

In this work, we do not further discuss cluster-scale parallelization (among Nodes in Figure 4.2). The CPU parallelization we'll describe in this section only leverages the parallelism among the multiple Sockets/Cores a single Node. As indicated by Figure 4.2, parallelization decisions at this level are analogous to decisions among the multiple GPUs in a single system, but we leave more detailed performance analysis of cluster-parallelization to future work.

4.4.2 Data-Parallelism and Geometric Decomposition

The computationally intense operations in MRI reconstructions contain nested data parallelism. In particular, operations such as Fourier and Wavelet transforms are performed over k-dimensional slices through the N-dimensional reconstruction volume, with $k < N$. In most cases the operations are performed for *all* k-dimensional (k-D) slices, providing another source of parallelism to be exploited for accelerating the reconstruction. The k-D operations themselves are parallelizable, but usually involve substantial synchronization and

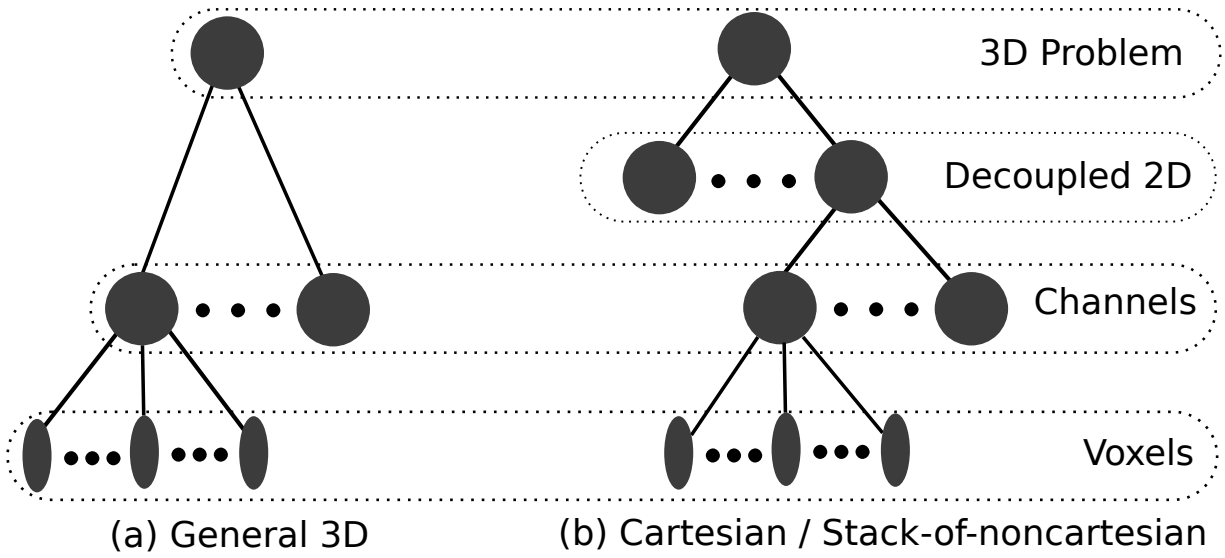

Figure 4.3: Hierarchical sources of parallelism in 3D MRI Reconstructions. For general reconstructions, operations are parallelizable both over channels and over image voxels. If there is a fully-sampled direction, for example the readout direction in Cartesian acquisitions, then decoupling along this dimension allows additional parallelization.

data-sharing. Whenever possible, it is very efficient to exploit this additional level of parallelism. For the purposes of software optimization, the size and shape of the N-dimensional (N-D) data are important. The Geometric Decomposition (GD) design pattern [58] discusses the design of parallel programs in which the data involved have geometric structure. GD suggests the parallelization should follow a division of the data that follows this structure, in order to achieve good caching and inter-thread communication behavior.

Recall from Figure 4.2 that modern processor architectures provide four levels at which to exploit parallelism. An efficient parallel implementation must decide at which levels of the processor hierarchy to exploit the levels of the nested parallelism in MRI reconstruction.

In volumetric MRI reconstructions, all of these operations are applied to the 4-D array representing the multi-channel 3D images. Figure 4.3 illustrates that the exploitable parallelism of operations over these arrays is two-level: operations like Fourier and Wavelet transforms applied to the individual channels' images exhibit massive voxel-wise parallelism and require frequent synchronization; but the transforms of the 4-32 channels can be performed independently and in parallel.

In cartesian acquisisions, the readout direction is never randomly subsampled. Similarly in stack-of-spirals or stack-of-radial acquisitions, the same non-cartesian sampling of $x - y$ slices is used for every z position. In these cases, the 3D reconstruction can be decoupled into independent 2D reconstructions for each undersampled slice. The resulting reconstruction

is unable to exploit cross-slice Wavelet-domain sparsity, and is potentially less noise-optimal than a fully 3-dimensional reconstruction. However, decoupling can provide a substantial performance benefit. Parallelizing over independent 2D reconstructions is very efficient, as the decoupled 2D reconstructions require no synchronization. Our Cuda ℓ1-SPIRiT solver is able to run multiple 2D problems simultaneously per GPU in batch mode. Large batch sizes require more GPU memory, but expose more parallelism and can more effectively utilize the GPU's compute resources.

4.4.3 Size-Dependence and Cache/Synchronization Trade-off

One can produce several functionally equivalent implementations by parallelizing at different levels of the hierarchy in Figure 4.2. These different implementations will produce identical results[1] but have very different performance characteristics. Moreover, the performance of a given implementation may differ substantially for different image matrix sizes and coil array sizes. In general, the optimal implementation is a trade-off between effective use of the cache/memory hierarchy and amortization of parallelization overheads.

For example, one may choose to exploit the voxel-wise parallelism in an operation only among the vector lanes within a single processor core. The implementation can then exploit parallelism over multiple channels and 2D slices over the multiple cores, sockets, and nodes in the system. Utilizing this additional parallelism will increase the memory footprint of the algorithm, as the working set of many 2D slices must be resident simultaneously. This consideration is particularly important for GPU systems which have substantially less DRAM capacity than CPU systems.

On the other hand, one may leverage voxel-wise parallelism among the multiple cores within a socket, the multiple sockets within the system, or among the multiple nodes. In doing so the implementation is able to exploit a larger slice of the system's processing and memory-system resources while simultaneously reducing memory footprint and working-set size. The favorable caching behavior of the smaller working set may result in a more efficient implementation. However, it is more expensive to synchronize the higher levels of the processing hierarchy. Furthermore, for problems with smaller matrix sizes, voxel-wise parallelism may be insufficient to fully saturate the processing resources at higher levels. Even when caching behavior is more favorable, this over-subscription of resources may degrade performance.

Which implementation provides better performance depends both on the size of the input data and on the size of the processor system. When the image matrix is very high-resolution (i.e. has a large number of voxels) or the processing system is relatively small (i.e. a small number of processor cores), then one can expect a high degree of efficiency from exploiting voxel-wise parallelism at higher levels of the hierarchy. If the image matrix is relatively

[1]Identical up to round-off differences in floating point arithmetic, which is not always associative or commutative

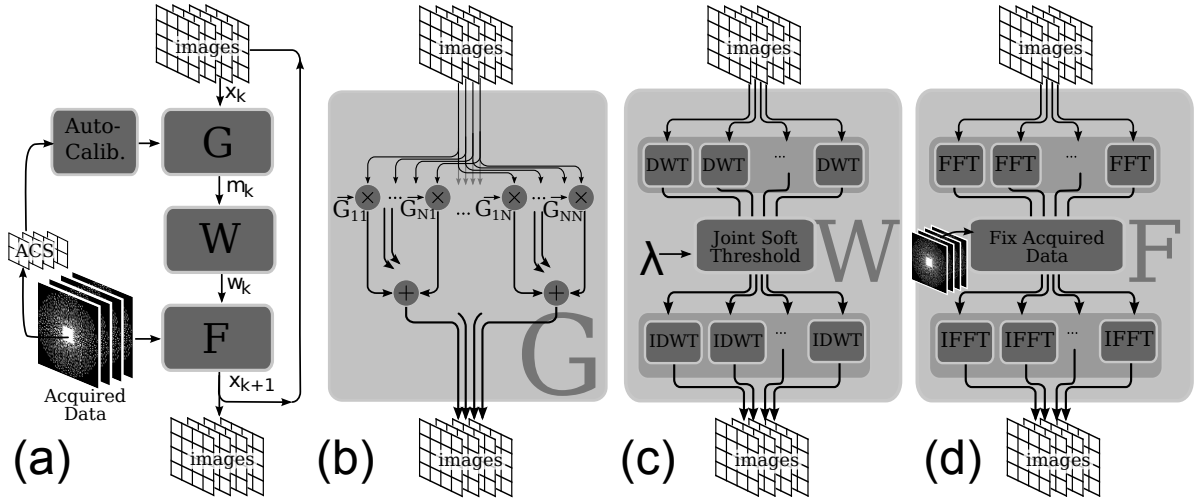

Figure 4.4: (a) Flowchart of the ℓ_1-SPIRiT POCS algorithm and the (b) SPIRiT, (c) Wavelet joint-threshold and (d) data-consistency projections

small or the processor system is very large, then one should expect that parallelism from the Channel and Decoupled-2D levels of Figure 4.3 is more important. As the number of processing cores per system and the amount of cache per core both continue to increase over time, we expect the latter case to become more common in the future.

4.4.4 Parallel Implementation of ℓ_1-SPIRiT

In the case of ℓ1-SPIRiT, there are four operations which dominate runtime: SPIRiT auto-calibration, Fourier transforms during the k-space consistency projection, Wavelet transforms during the joing soft-thresholding, and the image-domain implementation of SPIRiT interpolation. Figure 4.4 depicts the overall flow of the iterative reconstruction. Note that PI calibration must be performed only once per reconstruction, and is not part of the iterative loop.

SPIRiT Auto-Calibration Our ℓ1-SPIRiT implementation performs auto-calibration by fitting the SPIRiT consistency model to the densely sampled Auto-Calibration Signal (ACS), which requires solving a least-squares least-norm problem for each PI channel. Note that while we perform the POCS iterations over decoupled 2D slices, we perform calibration in full 3D. The SPIRiT interpolation kernels for the 2D problems are computed via an inverse Fourier transform in the readout direction.

As discussed in Section 4.5, the auto-calibration is computationaly dominated by two operations: the computation of a rank-k matrix product A^*A and a Cholesky factorization

$A = LL^*$, which itself is dominated by rank-k products. Numerical linear algebra libraries for both CPUs and GPUs are parallelized via an output-driven scheme that requires very little inter-thread synchronization. For example, when computing a matrix-matrix product $C = AB$ each element $c_{i,j}$ is computed as an inner product of a row a^i of A with a column b_j of B. All such products can be computed independently in parallel and are typically blocked to ensure favorable cache behavior.

The SPIRiT Operator in Image Space Figure 4.4 (b) illustrates the image-domain implementation of SPIRiT interpolation $\mathbf{G}x$. $\mathbf{G}x$ computes a matrix-vector multiplication per voxel – the n_c (# PI channels) length vector is composed of the voxels at a given location in all PI channels. For efficiency in the Wavelet and Fourier transforms, each channel must be stored contiguously. Thus the cross-channel vector for each voxel is non-contiguous. Our implementation of the interpolation streams through each channel in unit-stride to obviate inefficient long-stride accesses or costly data permutation. The image-domain implementation is substantially more efficient than the k-space implementation, which performs convolution rather than a multiplication. However, the image-domain representation of the convolution kernels requires a substantially larger memory footprint, as the compact k-space kernels must be zero-padded to the image size and Fourier transformed. Since SPIRiT's cross-coil interpolation is an all-to-all operation, there are n_c^2 such kernels. This limits the applicability of the image-domain SPIRiT interpolation when many large-coil-array 2D problems are in flight simultaneously. This limitation is more severe for the Cuda implementation than the OpenMP implementation, as GPUs typically have substantially less memory capacity than the host CPU system.

Enforcing Sparsity by Wavelet Thresholding Figure 4.4 (c) illustrates Wavelet Soft-thresholding. Similarly to the Fourier transforms, the Wavelet transforms are performed independently and in parallel for each channel. Our Wavelet transform implementation is a multi-level decomposition via a separable Daubechies 4-tap filter. Each level of the decomposition performs low-pass and high-pass filtering of both the rows and columns of the image. The number of levels of decomposition performed depends on the data size: we continue the wavelet decomposition until the approximation coefficients are smaller than the densely sampled auto-calibration region. Our OpenMP implementation performs the transform of a single 2D image in a single OpenMP thread, and parallelizes over channels 2D slices.

We will present performance results for two alternate GPU implementations of the Wavelet transform. The first parallelizes a 2D transform over multiple cores of the GPU, while the second is parallelized only over the vector lanes within a single core. The former is a finer-grained parallelization with a small working set per core, and permits an optimization that greatly improves memory system performance. As multiple cores share the transform for a single 2D transform, the per-core working set fits into the small l1-cache of the GPU.

Multiple Cuda thread blocks divide the work of the convolutions for each channel's image, and explicitly block the working data into the GPU's local store. Parallelism from multiple channels is exploited among multiple cores when a single 2D transform cannot saturate the entire GPU. The latter parallelization exploits all voxel-wise parallelism of a 2D transform within a single GPU core, and leverages the channel-wise and slice-wise parallelism across multiple cores. The working set of a single 2D image does not fit in the l1 cache of the GPU core and we cannot perform the explicit blocking performed in the previous case.

Enforcing k-space acquisition consistency Figure 4.4 (d) illustrates the operations performed in the k-space consistency projection. The runtime of this computation is dominated by the forward and inverse Fourier transforms. As the FFTs are performed independently for each channel, there is n_c-way communication-free parallelism in addition to the voxel-wise parallelism within the FFT of each channel. FFT libraries typically provide APIs to leverage the parallelism at either or both of these levels. The 2D FFTs of phase-encode slices in ℓ1-SPIRiT are efficiently parallelized over one or a few processor cores, and FFT libraries can very effectively utilize voxel-wise parallelism over vector lanes. We will present performance results for the GPU using both the `Plan2D` API, which executes a single 2D FFT at a time, and the `PlanMany` API which potentially executes many 2D FFTs simultaneously. The latter approach more easily saturates the GPU's compute resources, while the former approach is a more fine-grained parallelization with potentially more efficient cache-use.

4.5 $O(n^3)$ SPIRiT Calibration

As discussed in Section 4.3, our ℓ1-SPIRiT calibration solves a least-norm, least-squares (LNLS) fit to the fully-sampled auto-calibration signal (ACS) for each channel's interpolating coefficients. As each channel's set of coefficients interpolates from each of the the n_c channels, the matrix used to solve this system has $O(n_c)$ columns. Solving the n_c least-squares systems independently requires $O(n_c^4)$ time, and is prohibitively expensive. This section derives our algorithm for solving them in $O(n_c^3)$ time using a single matrix-matrix multiplication, single Cholesky factorization, and several inexpensive matrix-vector operations.

We construct a calibration data matrix A from the calibration data in the same manner as GRAPPA [37] and SPIRiT [57] calibration: each row of A is a window of the ACS the same size as the interpolation coefficients. This matrix is Toeplitz, and multiplication Ax by a vector $x \in \mathbb{C}^{n_c \cdot n_k}$ computes the SPIRiT interpolation: $y = \sum_{i=1}^{n_c} x_i * \mathrm{ACS}_i$.

The LNLS matrices for each channel differ only by a single column from A. In particular, there is a column of A that is identical to the ACS of each coil. Consider the coil corresponding to column i of A, and let b be that column. We define $N = A - be_i' - A$ with the i^{th} column zeroed out. We wish to solve $Nx = b$ in the least-norm least-squares sense, by solving $(N^*N + \varepsilon I)x := \tilde{N}x = N^*b$. The runtime of our efficient algorithm is dominated computing the product N^*N and computing the Cholesky factorization $LL^* = \tilde{N}$.

Our derivation begins by noting that:

$$\begin{aligned}
\tilde{N} &= N^*N + \varepsilon I \\
&= (A - be')^*(A - be') + \varepsilon I \\
&= A^*A + \varepsilon I - A^*be' - (be')^*A + (be')^*be' \\
&= A^*A + \varepsilon I - (A^*b)e' - e(A^*b)^* + (be')^*be' \\
&= A^*A + \varepsilon I - \tilde{b}e' - e\tilde{b}^*
\end{aligned}$$

Where we have defined $\tilde{b} = A^*b$ with entry i multiplied by $\frac{1}{2}$ to avoid adding eb^*be'. If we have a Cholesky factorization $LL^* = A^*A + \varepsilon I$:

$$\begin{aligned}
\tilde{N} &= LL^* - \tilde{b}e' - e\tilde{b}^* \\
&= LL^{-1}(LL^* - \tilde{b}e' - e\tilde{b}^*)L^{-*}L^* \\
&= L(I - L^{-1}\tilde{b}e'L^{-*} - L^{-1}e\tilde{b}^*L^{-*})L^* \\
&= L(I - (L^{-1}\tilde{b})(L^{-1}e)^* - (L^{-1}e)(L^{-1}\tilde{b})^*)L^* \\
&= L(I - \hat{b}\hat{e}^* - \hat{e}\hat{b}^*)L^*
\end{aligned}$$

Where we've defined $\hat{b} = L^{-1}\tilde{b} = L^{-1}A^*b$, and $\hat{e} = L^{-1}e$. These vectors can be computed with BLAS2 triangular solves and matrix-vector multiplications. In fact, we can aggregate the b's and e's from all parallel imaging channels into matrices and compute all $\hat{b}$'s and $\hat{e}$'s with highly efficient BLAS3 solves. Now, to solve the system of equations $\tilde{N}x = \tilde{b}$:

$$\begin{aligned}
x &= \tilde{N}^{-1}\tilde{b} \\
&= (L(I - \hat{b}\hat{e}^* - \hat{e}\hat{b}^*)L^*)^{-1}\tilde{b} \\
&= L^{-*}(I - \hat{b}\hat{e}^* - \hat{e}\hat{b}^*)^{-1}L^{-1}\tilde{b}
\end{aligned}$$

It remains to compute the inverse of $(I - \hat{b}\hat{e}^* - \hat{e}\hat{b}^*)$. We can define two matrices $\hat{B}, \hat{E} \in \mathbb{C}^{n\times 2}$, where $\hat{B} = -\left(\hat{b}, \hat{e}\right)$, and $\hat{E} = \left(\hat{e}, \hat{b}\right)$. Using the Sherman-Morrison-Woodbury identity:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

where in our case $A = C = I$, $U = \hat{B}$, and $V = \hat{E}^*$:

$$\begin{aligned}
(I - \hat{b}\hat{e}^* - \hat{e}\hat{b}^*)^{-1} &= (I + \hat{B}\hat{E}^*)^{-1} \\
&= I - I\hat{B}(I + \hat{E}^*I\hat{B})^{-1}\hat{E}^* \\
&= I - \hat{B}(I + \hat{E}^*\hat{B})^{-1}\hat{E}^*
\end{aligned}$$

Note that $I + \hat{E}^*\hat{B}$ is a 2×2 matrix that is very inexpensive to invert. Thus we have our

Dataset	n_x	n_y	n_z	n_c
A	192	256	58	8, 16, 32
B	192	256	102	8, 16, 32
C	192	256	152	8, 16, 32
D	192	256	190	8, 16, 32
E	320	260	250	8, 16, 32
F	320	232	252	8, 16, 32

Table 4.1: Table of dataset sizes for which we present performance data. n_x is the length of a readout, n_y and n_z are the size of the image matrix in the phase-encoded dimensions, and n_c is the number of channels in the acquired data. Performance of SPIRiT is very sensitive to the number of channels, so we present runtimes for the raw 32-channel data as well as coil-compressed 8- and 16- channel data.

final algorithm:

$$\begin{aligned}
L &\leftarrow \text{chol}(A^*A + \varepsilon I) \\
\hat{b} &\leftarrow L^{-1}A^*b \\
\hat{e} &\leftarrow L^{-1}e \\
\hat{B} &\leftarrow -\left(\hat{b}, \hat{e}\right) \\
\hat{E} &\leftarrow \left(\hat{e}, \hat{b}\right) \\
x &\leftarrow L^{-*}(I - \hat{B}(I + \hat{E}^*\hat{B})^{-1}\hat{E}^*)\hat{b} \\
x &\leftarrow L^{-*}(I - \hat{B}(I + \hat{E}^*\hat{B})^{-1}\hat{E}^*)L^{-1}\tilde{b}
\end{aligned}$$

4.6 Methods

Section 4.7 presents performance results for a representative sample of datasets from our clinical application. We present runtimes for a constant number of POCS iterations only, so the runtime depends only on the size of the input matrix. In particular, our reported runtimes do not depend on convergence rates or the amount of scan acceleration. We present performance results for six datasets, whose sizes are listed in Table 4.1.

We present several performance metrics of interest. First, we shall discuss the end-to-end runtime of our reconstruction to demonstrate the amount of wall-clock time the radiologist must wait from the end of the scan until the images are available. This includes the PI calibration, the POCS solver, and miscellaneous supporting operations. To avoid data-dependent performance differences due to differing convergence rates, we present runtime for a constant (50) number of POCS iterations.

To demonstrate the effectiveness of our $O(n^3)$ calibration algorithm, we compare its runtime to that of the "obvious" implementation which uses ACML's implementation of the Lapack routine `cposv` to solve each coil's calibration independently. The runtime of calibration does not depend on the final matrix size, but rather on the number of PI channels and the number of auto-calibration readouts. We present runtimes for calibrating $7 \times 7 \times 7$ kernels averaged over a variety of ACS sizes.

We also present per-iteration runtime and execution profile of the POCS solver for several different parallelizations, including both CPU and GPU implementations. The per-iteration runtime does not depend on the readout length or the rate of convergence. Since we decouple along the readout dimension, POCS runtime is simply linear in n_x. Presenting per-iteration runtime allows direct comparison of the different performance bottlenecks of our multiple implementations.

Additionally, we explore the dependence of performance on data-size by comparing two alternate implementations parallelized for the GPU. The first exploits voxel-wise parallelism and channel-wise parallelism at the Socket-level from Figure 4.2 and does not exploit Decoupled-2D parallelism. This implementation primarily synchronizes by ending Cuda grid launches, incurring substantial overhead. However, the reduced working-set size increases the likelihood of favorable cache behavior, and enables further caching optimizations as described in Section 4.4.4. Fourier transforms are performed via the 2D API, which expresses a single parallel FFT per grid launch. Fermi-class GPUs are able to execute multiple grid launches simultaneously, thus this implementation expresses channel-wise parallelism as well. The second implementation exploits voxel-wise parallelism only within a core of the GPU, and maps the channel-wise and Decoupled-2D parallelism at the Socket-level. This implementation is able to use the more efficient within-core synchronization mechanisms, but it has a larger working set per core and thus cannot as effectively exploit the GPU's caches. It also launches more work simultaneously in each GPU grid launch than does the first implementation, and can more effectively amortize parallelization overheads. Fourier transforms are performed via the "Plan Many" API, which expresses the parallelism from all FFTs across all channels and all slices simultaneously.

All performance data shown were collected on our dual-socket $\times$ six-core Intel Xeon X5650 @2.67GHz system with four Nvidia GTX580s in PCI-Express slots. The system has 64GB of CPU DRAM, and 3GB of GPU DRAM per card (total 12 GB). We leverage Nvidia's Cuda [62] extensions to C/C++ to leverage massively parallel GPGPU processors, and OpenMP [2] to leverage multi-Core parallelism on the system's CPUs. Additionally, multiple OpenMP threads are used to manage the interaction with the system's multiple discrete GPUs in parallel. We leverage freely available high-performance libraries for standard operations: ACML [3] for linear system solvers and matrix factorizations, FFTW [4] and CUFFT [5]

[2] `http://www.openmp.org`
[3] `http://www.amd.com/acml`
[4] `http://www.fftw.org`
[5] `http://developer.nvidia.com/cuda-toolkit-40`

for Fourier transforms.

4.7 Performance Results

Figures 4.5-4.10 present performance data for our parallelized ℓ1-SPIRiT implementations. Figure 4.5 shows stacked bar charts indicating the amount of wall-clock time spent during reconstruction of the six clinical datasets, whose sizes are listed in Section 4.6. The POCS solver is run with a single 2D slice in flight per GPU. This configuration minimizes memory footprint and is most portable across the widest variety of Cuda-capable GPUs. It is the default in our implementation. The stacked bars in Figure 4.5 represent:

- **3D Calibration**: The SPIRiT calibration that computes the SPIRiT **G** operator from the ACS data as described in Section 4.3. Figure 4.9 presents more analysis of this portion.
- **POCS**: The per-slice 2D data are reconstructed via the algorithm described in Figure 4.1. Figure 4.6 presents a more detailed analysis of the runtime of this portion.
- **other:** Several other steps must also be performed during the reconstruction, including data permutation and IFFT of the readout dimension.

Figures 4.6 and 4.7 show the contribution of each individual algorithmic step to the overall runtime of a single iteration of the 2D POCS solver. In Figure 4.6, the solver is parallelized so that a single 2D problem is in-flight per GPU. In Figure 4.7, a single 2D problem is in flight per CPU core. The stacked bars in Figure 4.6 and Figure 4.7 are:

- **FFT**: The Fourier transforms performed during the k-space consistency projection.
- **SPIRiT Gx**: Our image-domain implementation of the SPIRiT interpolation, which performs a matrix-vector multiplication per voxel.
- **Wavelet**: The Wavelet transforms performed during wavelet soft-thresholding.
- **other**: Other operations that contribute to runtime include data movement, joint soft-thresholding, and the non-Fourier components of the k-space projection.

Figure 4.8 compares the runtime of the Parallel GPU and CPU POCS implementations to the runtime of a sequential C++ implementation. The reported speedup is computed as the ratio of the sequential runtime to the parallel runtime.

Figure 4.9 demonstrates the runtime of the efficient Cholesky-based SPIRiT calibration algorithm described in Section 4.5. The left graph compares the runtime of of our efficient $O(n^3)$ calibration to the naïve $O(n^4)$ algorithm. The right plot shows what fraction of the efficient algorithm's runtime is spent in the matrix-matrix multiplication, the Cholesky factorization, and the other BLAS2 matrix-vector operations.

4.8 Discussion

Figures 4.5 and 4.6 present performance details for the most portable GPU implementation of the POCS solver which runs a single 2D slice per GPU . As shown in Figure 4.5, our GPU-parallelized implementation reconstructs datasets A-D at 8 channels in less than 30 seconds, and requires about 1 minute for the larger E and F datasets. Similarly, our reconstruction runtime is 1-2 minutes for all but the 32-channel E and F data, which require about 5 minutes. Due to the $O(n_c^3)$ complexity of calibration, calibration requires a substantially higher fraction of runtime for the 32-channel reconstructions, compared to the 8 an 16-channel reconstructions. Similarly, Figure 4.6 shows that the $O(n_c^2)$ SPIRiT interpolation is a substantial fraction of the 32-channel POCS runtimes as well. Figures 4.5 and 4.6 demonstrate another important trend of the performance of this GPU implementation. Although dataset D is 4× larger than dataset A, the 8-channel GPU POCS runtimes differ only by about 10%. The trend is clearest in the performance of the Fourier and Wavelet transforms, whose runtime is approximately the same for datasets A-D. This is indicative of the inefficiency of the CUFFT library's *Plan2D* API for these small matrix sizes. In a moment we'll discuss how an alternate parallelization strategy can substantially improve efficiency for these operations.

Figures 4.7 presents the averaged per-iteration execution profile of the OpenMP-parallelized CPU POCS solver, which uses an `#pragma omp for` to perform a single 2D slice's reconstruction at a time per thread. The relative runtimes of the Fourier and Wavelet transforms are more balanced in the CPU case. In particular, the CPU implementation does not suffer from low FFT performance for the small data sizes. The FFT is run sequentially within a single OpenMP thread, and it incurs no synchronization costs or parallelization overhead.

Figure 4.8 presents the speedup of the multi-GPU solver and the multicore CPU solver over a sequential C++ implementation. Note that the 4-GPU implementation is only about 33% faster than the 12-CPU implementation for the smallest data size (dataset A at 8 channels), while for the larger reconstructions the GPU implementation is 5×–7× faster. The OpenMP parallelization consistently gives 10×–12× speedup over sequential C++, while the multi-GPU parallelization provides 30×–60× speedup for most datasets.

Figure 4.9 demonstrates the enormous runtime improvement in SPIRiT calibration due to our Cholesky-based algorithm described in Section 4.3 and derived in Section 4.5. The runtime of our calibration algorithm is dominated by a single large matrix-matrix multiplication, Cholesky decomposition, and various BLAS2 (Matrix-vector) operations. For 8 channel reconstructions, the $O(n^3)$ algorithm is faster by $2-3\times$, while it is 10× faster for 32 channel data. In absolute terms, 8-channel calibrations require less than 10 seconds when computed via either algorithm. However, 32 channel calibrations run in 1–2 minutes via the Cholesky-based algorithm, while the $O(n^4)$ algorithm runs for over 15 minutes.

Figure 4.10 provides a comparison of alternate parallelizations of the POCS solver and the dependence of performance on data size. The "No Batching" implementation exploits the voxel-wise and channel-wise parallelism within a single 2D problem per GPU Socket. The

remaining bars batch multiple 2D slices per GPU Socket. The top bar graph shows runtimes for a small 256×58 image matrix, and the bottom graph shows runtimes for a moderately sized 232×252 matrix. Both reconstructions were performed after coil-compression to 8 channels.

Fourier transforms in the No Batching implementation are particularly inefficient for the small data size. The 256×58 transforms for the 8 channels are unable to saturate the GPU. The "Batched 1x" bar uses the `PlanMany` API rather than the `Plan2D` API. This change improves FFT performance, demonstrating the relative ineffectiveness of the GPU's ability to execute multiple grid launches simultaneously. Performance continues to improve as we increase the number of slices simultaneously in-flight, and the FFTs of the small matrix are approximately $5\times$ faster when batched $32\times$. However, for the larger 232×252 dataset, $32\times$ batching achieves performance approximately equal to the non-batched implementation. That the $1\times$ batched performance is worse than the non-batched performance likely indicates that the larger FFT is able to exploit multiple GPU cores.

Our Wavelet transforms are always more efficient without batching, as the implementation is able to exploit the GPU's small scratchpad caches (Cuda `__shared__` memory) as described in Section 4.4.4. The wavelet transform performs convolution of the low-pass and high-pass filters with both the rows and the columns of the image. Our images are stored in column-major ordering, and thus we expect good caching behavior for the column-wise convolutions. However, the row-wise convolutions access the images in non-unit-stride without our scratchpad-based optimizations. Comparing the runtimes of the "No Batching" and "Batched 1x" Wavelet implementations in Figure 4.10 shows that our cache optimization can improve performance by $3\times$–$4\times$. This is a sensible result, as we use 4-tap filters and each pixel is accessed 4 times per convolution. The cache optimization reduces the cost to a single DRAM access and 3 cached accesses.

Note that performance could be improved by choosing different parallelization strategies for the various operations. In particular, the best performance would be achieved by using a batched implementation of the Fourier transforms, while using the un-batched implementation of the Wavelet transforms. Such an implementation would still require the larger DRAM footprint of the batched implementation, as multiple 2D slices must be resident in GPU DRAM simultaneously. However it could achieve high efficiency in the Wavelet transform via the caching optimization, and also in the Fourier transforms via higher processor utilization. Although our current implementation does not support this hybrid configuration, the rightmost graph in Figure 4.10 shows that it could perform up to $2\times$ faster for the 256×58 dataset. Moore's Law scaling will result in higher core counts in future architectures. Per Gustafson's law [39], efficient utilization of future architectures will require increased problem size. In our context, we can increase problem size via larger batch sizes. Per-batch reconstruction time will remain constant, but total reconstruction time will be inversely proportional to batch size. Thus batching potentially provides linear performance scaling with increased core counts.

We also anticipate that as Moore's Law scaling will result in in higher numbers of pro-

cessor cores per socket in the future, this type of parallelization may become increasingly important: relative to larger processors, the majority of clinical datasets will appear smaller.

4.9 Image Quality

We present more comprehensive evaluation of image quality in prior works [81]. Figure 4.11 presents a clinical case demonstrating the image quality advantage that our reconstruction can provide. Our 3-Dimensional Compressed Sensing pulse sequence is a modified 3DFT spoiled gradient-echo (SPGR) sequence which undersamples in both of the phase-encoded dimensions (y) and (z). Acquisitions are highly accelerated, with 4×–8× undersampling of phase encodes. Our clinical imaging is performed using 3T and 1.5T GE systems with a with 32-channel pediatric torso coil. Typical accelerated scan times are 10–15 seconds. We perform reconstructions both via our ℓ_1-SPIRiT implementation and the GE Autocalibrating Reconstruction for Cartesian imaging (ARC) [6], which is capable of reconstructing arbitrary Cartesian k-space subsamplings. Typical ARC reconstruction times are 30-60 seconds. In some cases, we perform partial k-space acquisition in the readout direction. The ℓ_1-SPIRiT solver is still able to decouple the 2D reconstructions as described in Section 4.4.2, using only the acquired portion of the readout. Subsequently, we perform Homodyne reconstruction [61] to estimate the missing portion of readout. Although we could achieve potentially higher image quality by incorporating the conjugate-symmetry assumption as a projection in the POCS algorithm, our current approach is effective at preventing blur and phase from appearing in the final images [55]. Our reconstruction is performed on-line with coil compression, producing sub-minute runtimes for matrix sizes typically acquired in the clinic. Total latency from scan completion to image availability is 2–5 minutes, 20-70 seconds of which are the ℓ_1-SPIRiT solver. The remainder of the reconstruction time is spent performing Grad-Warp [35] and Homodyne processing steps on the CPUs, in addition to file transfers between the scanner and our reconstruction system.

4.10 Conclusion

We have presented ℓ_1-SPIRiT, a compressive sensing extension to the SPIRiT parallel imaging reconstruction. Our implementation of ℓ_1-SPIRiT for GPGPUs and multi-core CPUs achieves clinically feasible sub-minute runtimes for highly accelerated, high-resolution scans. We discussed in general terms the software implementation and optimization decisions that contribute to our fast runtimes, and how they apply for the individual operations in ℓ_1-SPIRiT. We presented performance data for both CPU and GPU systems, and discussed how a hybrid parallelization may achieve faster runtimes. Finally, we present an image quality comparison with a competing non-iterative Parallel Imaging reconstruction approach.

In the spirit of reproducible research, the software described in this chapter is available

at: `http://www.eecs.berkeley.edu/~mlustig/Software.html`.

4.11 Acknowledgments

Many thanks to our collaborators in this work: Marcus Alley, James Demmel, and Shreyas Vasanawala

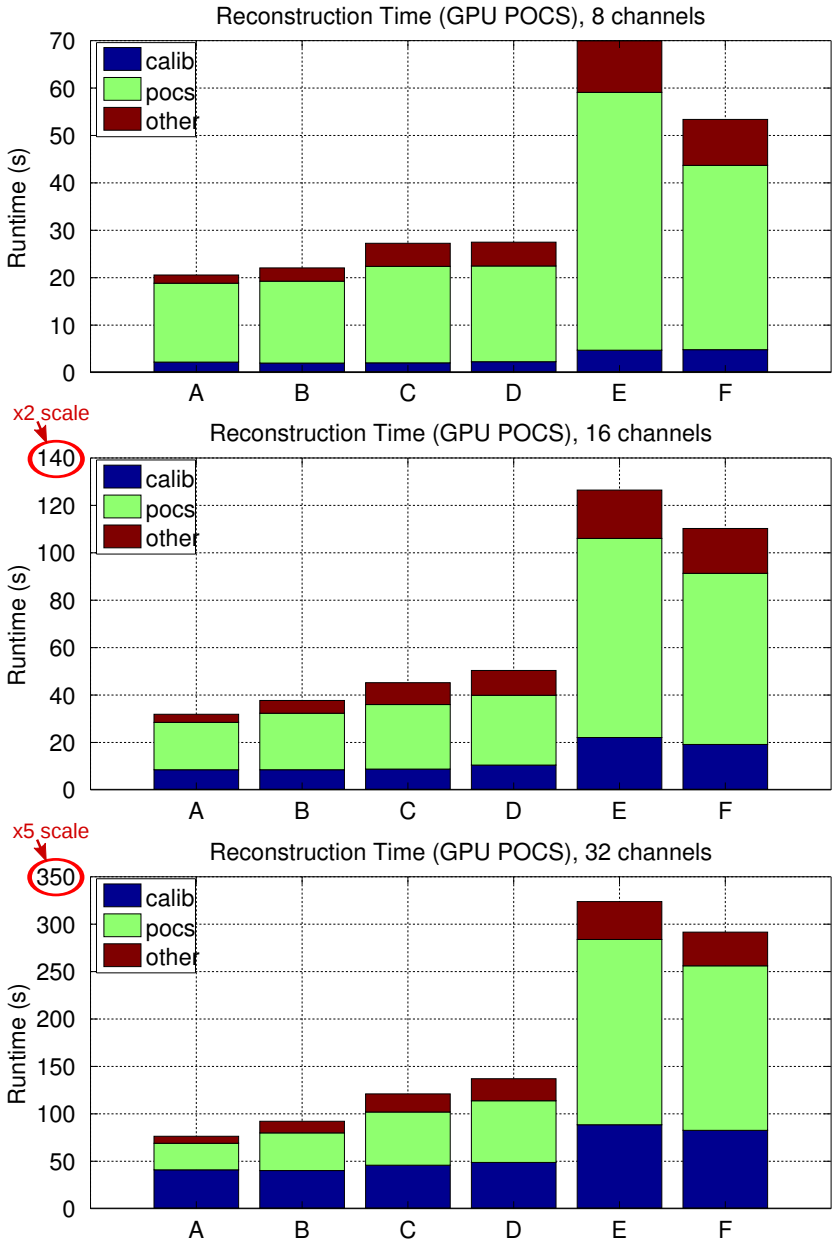

Figure 4.5: Reconstruction runtimes of our ℓ_1-SPIRiT solver for 8-, 16-, and 32-channel reconstructions using the efficient Cholesky-based calibration and the multi-GPU POCS solver.

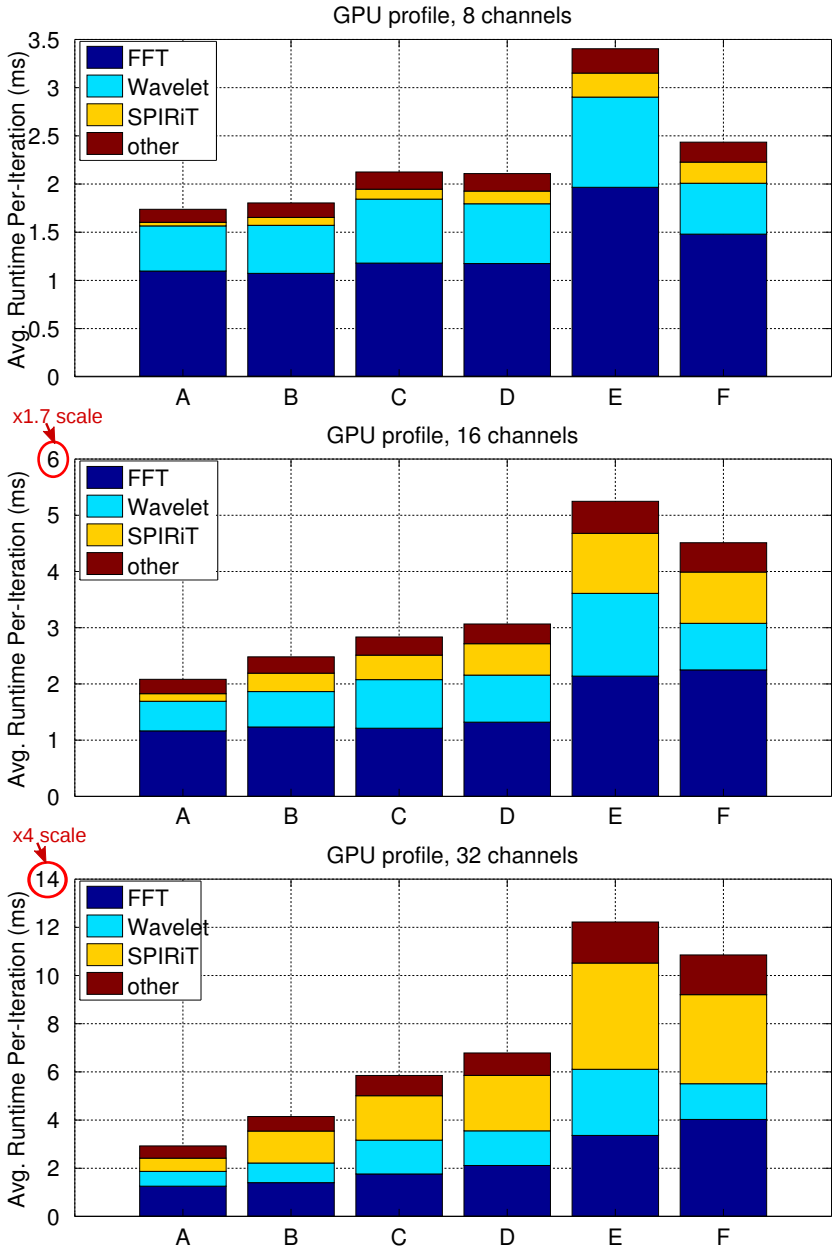

Figure 4.6: Per-iteration runtime and execution profile of the GPU POCS solver.

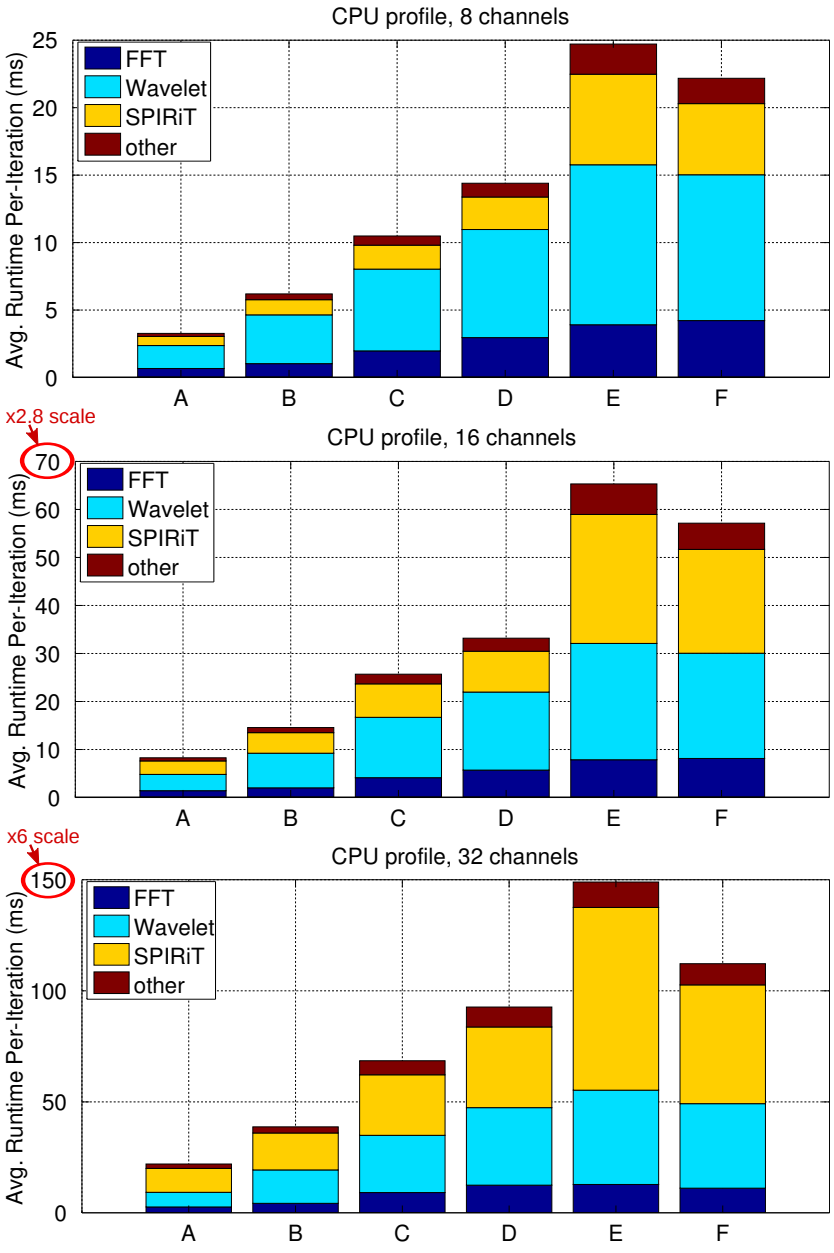

Figure 4.7: Per-iteration runtime and execution profile of the multi-core CPU POCS solver.

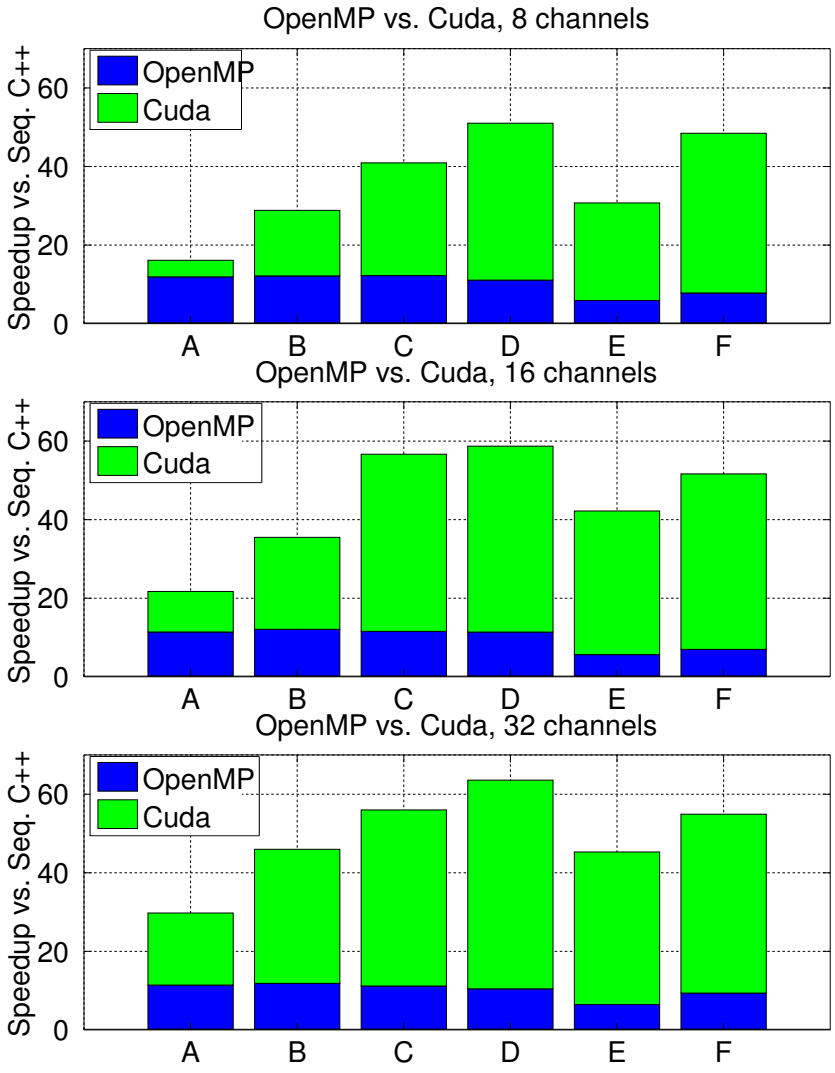

Figure 4.8: Speedup of parallel CPU and GPU implementations of the POCS solver over the sequential C++ runtime.

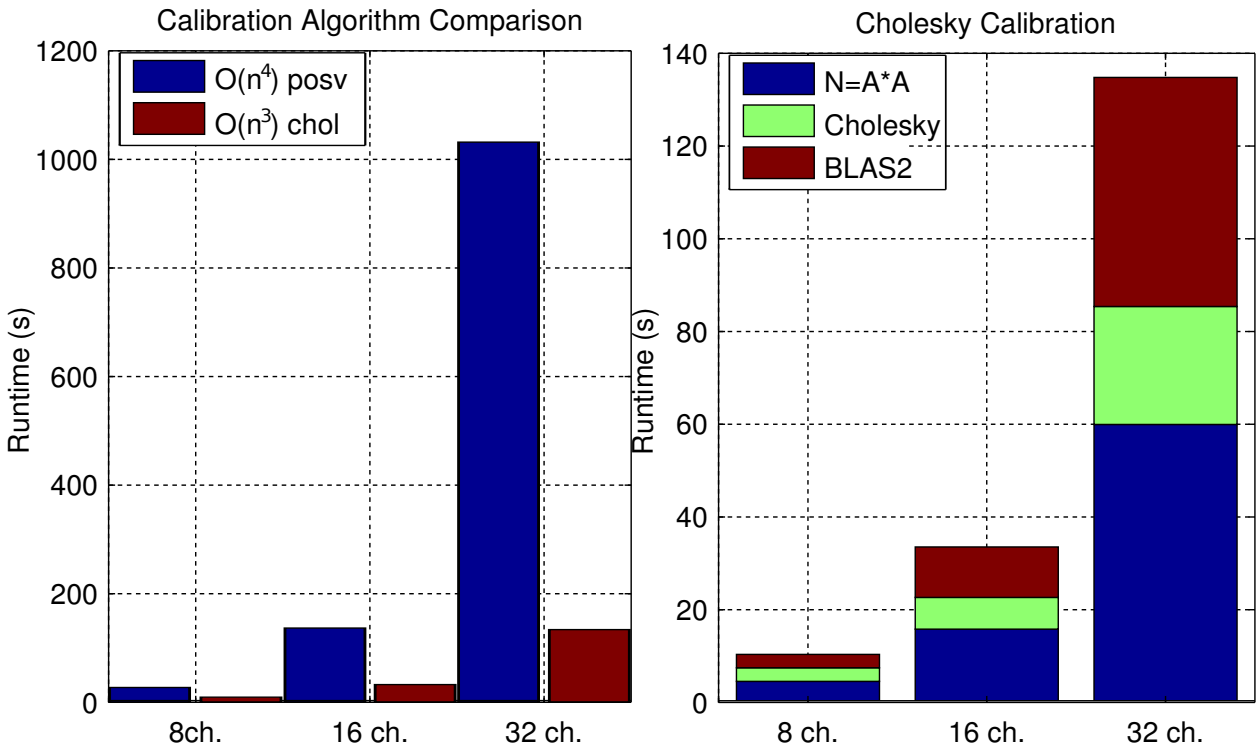

Figure 4.9: 3D SPIRiT Calibration runtimes, averaged over a variety of auto-calibration region sizes. Calibration is always performed on the CPUs via optimized library routines, using all available threads

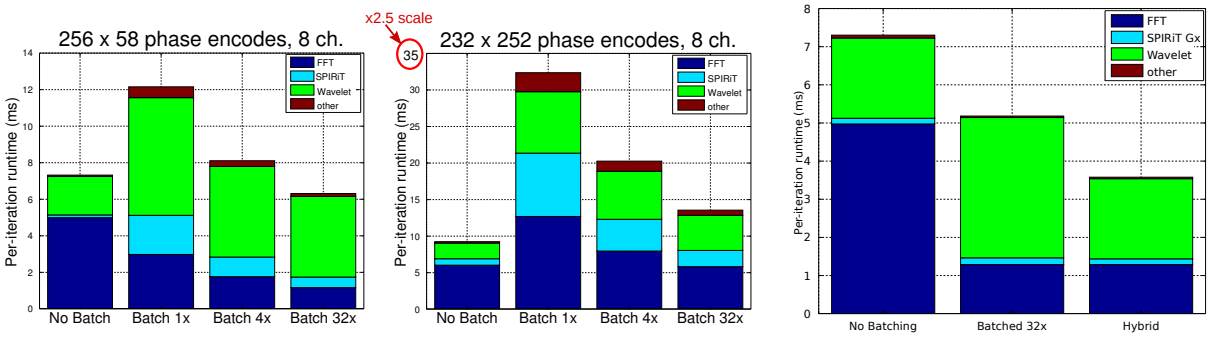

Figure 4.10: (left and middle) Data Size dependence of performance and comparison of alternate parallelizations of the POCS solver. (right) Performance achievable by a hybrid parallelization of the POCS solver on the 256×58 dataset.

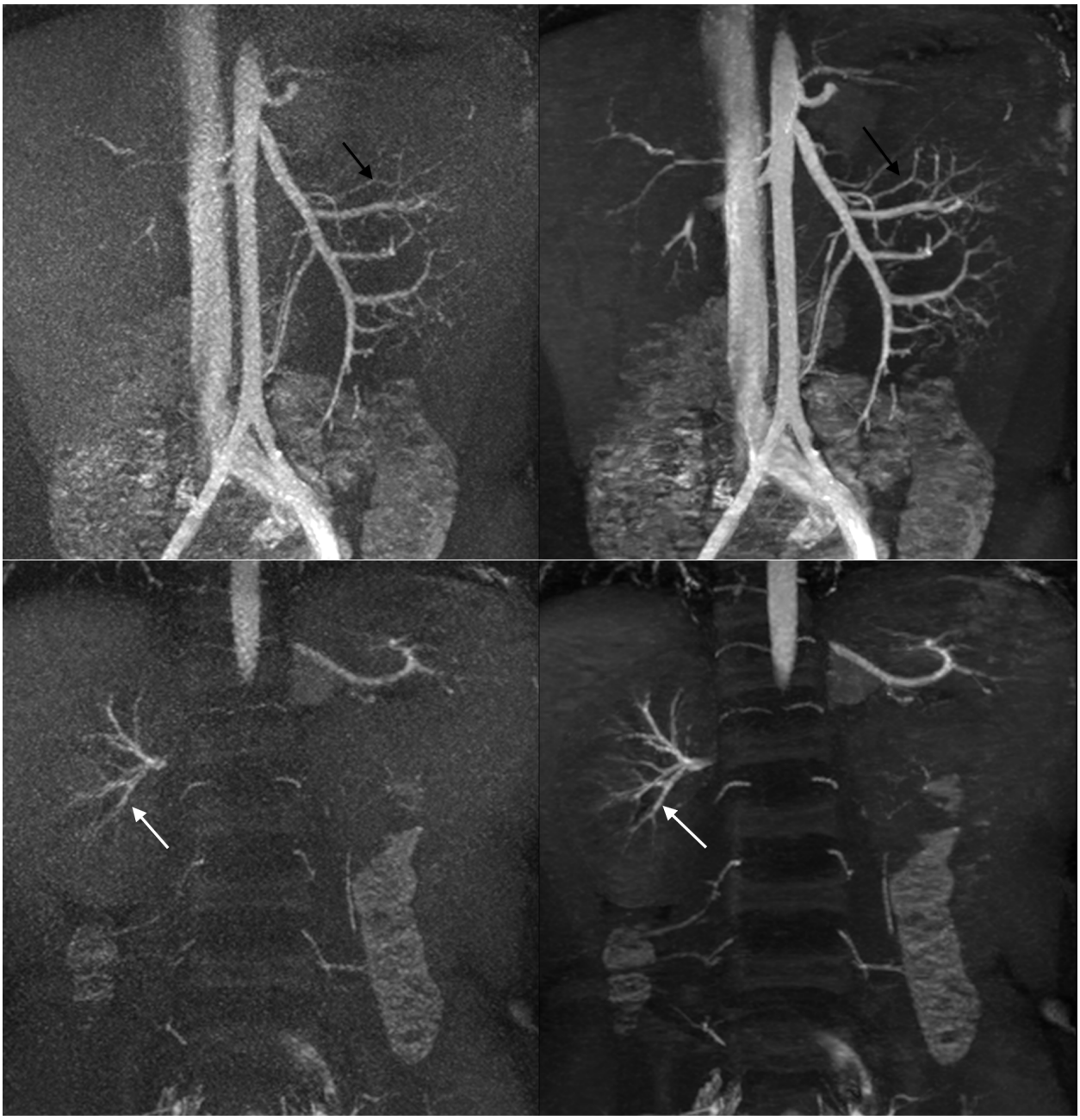

Figure 4.11: Image quality comparison of GE Product ARC (Autocalibrating Reconstruction for Cartesian imaging) [6] reconstruction (left images) with our ℓ_1-SPIRiT reconstruction (right images). Both reconstructions use the same subsampled data, and require similar runtimes. These MRA images of a 5 year old patient were acquired with the 32-channel pediatric torso coil, have FOV 28 cm, matrix size 320 $\times$ 320, slice thickness 0.8 mm, and were acquired with 7.2$\times$ acceleration, via undersampling 3.6$\times$ in the y-direction and 2$\times$ in the z-direction. The pulse sequence used a 15 degree flip angle and a TR of 3.9 ms. The ℓ_1-SPIRiT reconstruction shows enhanced detail in the mesenteric vessels in the top images, and and the renal vessels in bottom images.

Chapter 5

nuFFTWC: The Fastest Non-Uniform FFT from the West Coast

5.1 Introduction

From the discussion of the Supermarionation architecture in Chapter 3, it is clear that the productive development of high-performance MRI reconstructions depends crucially on the availability of highly optimized libraries of commonly used computations. The variety of MRI applications necessitates that reconstructions will differ greatly in the overall sequence of computations performed and in the iterative algorithms used to solve MRI inverse problems. However, due to the fundamental commonalities of these applications, for example sampling in k-space and Wavelet-domain sparsity, much of the computational expense of MRI reconstructions lies in a small set of algorithms. This chapter describes in depth the performance optimization of a non-unform fast Fourier transform algorithm known as Gridding in MRI reconstruction literature. While Chapter 4 described the implementation of a reconstruction algorithm relying on multiple highly optimized operations, this chapter provides more insight into the techniques used to achieve efficiency in the low-level operations that contribute most to the computational expense of iterative reconstructions.

In Magnetic Resonance Imaging (MRI), data are acquired as samples in the Fourier domain (k-space). The sample locations are determined by the sequence of radio frequency pulses and magnetic field gradients applied during imaging. Reconstruction of images from k-space data is typically posed as a Fourier-inversion problem, and require computation of the inverse Fourier transform of the sampled data. Most clinical imaging is performed via pulse sequences that sample k-space at equispaced locations, and images can be reconstructed via the well-known Fast Fourier Transform (FFT). The FFT of N samples requires $O(N \log N)$ arithmetic operations, whereas a direct computation of the Discrete Fourier Transform (DFT) would require $O(N^2)$ operations. Computational feasibility of MRI reconstruction depends heavily on the use of FFTs, especially in iterative reconstructions where

the Fourier transform must be computed at least once in each iteration. When the k-space sample locations do not lie on a Cartesian grid, a non-Uniform Fast Fourier Transform (nuFFT) is used instead of a standard FFT. The Gridding algorithm (with density compensation) is an implementation used in MRI [10] to approximate an inverse Fourier transform of non-equispaced k-space samplings. Due to the efficiency and image quality achievable by Gridding, it is the most commonly used algorithm for non-Cartesian reconstructions. As we'll discuss in detail in Section 5.2, the nuFFT approximates the NDFT by re-sampling the data at equispaced locations on an over-sampled grid. The FFT is then used to compute the Fourier transform of the grid. Re-sampling k-space produces aliasing in the image domain, and over-sampling increases the field of view (FOV) of the reconstructed image. The interpolation kernel and degree of over-sampling are chosen so that aliasing artifacts appear in the extended FOV, rather than the central region containing the reconstructed image.

The accuracy and image quality of the nuFFT in MRI applications has been well-studied in prior literature. Jackson *et al.* [44] evaluate a number of different interpolation kernels to be used during the re-sampling, determining that Kaiser-Bessel windows (with appropriate free-parameter selection) achieve very close to optimal reconstruction quality. Fessler *et al.* [31] propose a min-max criterion for nuFFT optimality that provides a closed-form solution for the optimal re-sampling interpolation kernel. Their kernel exhibits several desirable properties, including several options for partial precomputation to reduce memory footprint and runtime. Beatty *et al.* [7] discuss the selection of nuFFT parameters to substantially reduce memory footprint and runtime, while maintaining clinical-quality images. Additionally they propose the *maximum aliasing amplitude* metric, which provides a data-independent measure of nuFFT accuracy.

The accuracy of the nuFFT is well-understood, but its efficient implementation on modern parallel processors has been a topic of much recent study. These works focus on the resampling interpolation, and differ primarily in the amount of precomputation they perform. Sørensen *et al.*[73] describe a clever parallelization of resampling that pre-computes the set of non-equispaced samples nearby each equispaced grid location in order to obviate synchronization among parallel threads of execution. As their work was implemented on an early General-Purpose Graphics Processing Unit (GPGPU), this precomputation step was necessary for correctness. More recently Obeid *et al.* [64] describe a spatial "binning" step which sorts the non-Cartesian samples by their k-space coordinates, also to obviate inter-thread synchronization in a GPGPU implementation. Their approach permutes the non-equispaced samples, and potentially improves cache performance as well. In the same conference proceedings, Nam *et al.* [1] present a substantial GPGPU speedup for gridding of a 3D radial trajectory.

We rely heavily on the existence of highly optimized libraries for the FFT and sparse matrix operations, which have been well-studied in the scientific computing literature. Modern processor architectures are highly complex, and it is difficult to model performance accurately. For FFTs and sparse matrix operations, empirical search over a space of functionally equivalent implementations has proven a fruitful approach for designing high-performance

libraries. These "self-tuning" libraries determine the optimal set of optimization parameters via a combination of benchmarking and heuristics. The ATLAS [84] (Automatically Tuned Linear Algebra Software) library is a self-tuning library for the Basic Linear Algebra Subroutines (BLAS) used heavily in most numerical applications. The well-known Fastest Fourier Transform in the West[1] (FFTW) [33] generates high-performance implementations of FFTs by evaluating alternative Cooley-Tukey factorizations, among a number of other implementation parameters. The Optimized Sparse Kernel Interface (OSKI) [83] is a self-tuning library for several common sparse matrix operations, including the matrix-vector multiplication we use in this work. The NFFT 3.0 library [46] is a very flexible implementation of the non-uniform FFT, suitable for use in a variety of numerical computing applications, including MRI reconstruction [50, 28]. The NFFT library provides a split plan/execute interface, similar to that of the FFTW library [33]. NFFT's planner provides a variety of precomputation options, including full precomputation of the matrix we define as $\mathbf{\Gamma}$ in Section 5.2, although their planner performs no empirically-driven performance optimization.

FFTW was the first library to perform performance optimization in a separate "planner" routine. Transforms are computed via a separate "execute" routine. The existence of a planning phase greatly simplifies the design and use of high-performance libraries. The optimizations that auto-tuning libraries perform require specification of some properties of thr operation to be computed, such as FFT size or the locations of nonzeros in a sparse matrix. The planner phase takes as input only the specification of the operation, and returns an opaque "plan" object. Applications can re-use the same plan to perform multiple transforms, and amortize any time spent determining the optimal plan.

In this work, we propose an implementation of the Gridding nuFFT that uses a self-tuning optimization strategy to ensure high performance on modern parallel processing platforms. Our strategy permits a split plan/execute programming interface, similar to those used by other self-tuning libraries. The transform is specified to the planner as a pattern of non-uniform samples, a final image matrix size, and a desired level of Fourier transform approximation accuracy. We note that if appropriate data structures are precomputed, all of the expensive computations in the nuFFT can be performed via extant highly optimized libraries. Much of the efficiency of our implementation is due to these libraries. We evaluate our approach using a variety of transforms drawn from the application of the Gridding nuFFT in MR Image reconstruction.

5.2 Gridding nuFFT Algorithm

Let $f(x)$ be a signal that we have sampled at M locations $\{x_m\}$. The signal is often complex-valued, and our index space is usually multi-dimensional. We denote by $F(k)$ the Fourier transform of f. The Fourier transform of f evaluated at a spatial frequency k_n is defined as

[1]The title of our work, nuFFTWC, is a tribute to FFTW

the inner product with the appropriate Fourier basis function:

$$F_n = \frac{1}{\sqrt{M}} \sum_{m=1}^{M} f_m e^{-2\pi\imath(x_m \cdot k_n)} \tag{5.1}$$

where we define $f_m \triangleq f(x_m)$, $F_n \triangleq F(k_n)$, and $\imath = \sqrt{-1}$ as the imaginary number. Applications typically require evaluation of the Fourier transform at a set of N different frequencies $\{k_n\}$. In general, $M \neq N$ and we cannot define a unique inverse of the linear transformation described by Equation (5.1). However we can easily define its adjoint operator as

$$f_m = \frac{1}{\sqrt{M}} \sum_{n=1}^{N} F_n e^{2\pi\imath(x_m \cdot k_n)}. \tag{5.2}$$

Normalization by $\sqrt{M}$ is motivated by the case of $M = N$ and equispaced sampling. In this case, the Fourier transform is the well-known Discrete Fourier Transform (DFT) and the adjoint is an inverse DFT. Our work is motivated by the Fourier inverse problems that arise in MRI reconstruction. Computation of the adjoint is required by some algorithms for the solution of these problems, for example the bi-Conjugate Gradient (BiCG) method [5] for solving non-symmetric linear systems. Historically in MRI literature, the term "Gridding" has been used to describe the reconstruction of images from non-equispaced Fourier-domain samples (i.e. the adjoint Fourier transform), and the terms "re-Gridding" or "inverse-Gridding" have been used to describe the computation of non-equispaced Fourier coefficients.

When both the spatial samples $\{x_m\}$ and the frequency samples $\{k_n\}$ lie at evenly-spaced positions, the well-known Fast Fourier Transform (FFT) algorithm can evaluate the Fourier transform very efficiently. In many applications, including the non-Cartesian MRI reconstructions that motivate our work, at least one of the the sample-point sets $\{x_m\}$ or $\{k_n\}$ is not equispaced. The discrete Fourier transform in this context is commonly known as a Non-uniform DFT (NDFT), and no log-linear algorithm exists for its exact evaluation. Computing the NDFT directly requires $O(MN)$ arithmetic operations, and for many applications is unfeasibly expensive.

Instead, these applications rely on a class of fast approximation algorithms referred to as Non-Uniform Fast Fourier Transforms (nuFFT). Most nuFFT algorithms re-sample the data onto an equispaced grid, enabling the use of the FFT to compute the Fourier transform. If both sets $\{x_m\}$ and $\{k_n\}$ are non-equispaced, then the output of the FFT must be re-sampled as well. Resampling is defined as the convolution of $f(k)$ with some real-valued interpolation kernel $g(k)$:

$$(f * g)(k) = \int_{||k-\kappa||_2 < \frac{1}{2}W} f(\kappa) g(k-\kappa) d\kappa. \tag{5.3}$$

For computational feasibility, the convolution kernel is defined to be of finite width W – i.e. $g(k) = 0$ if $||k||_2 \geq \frac{1}{2}W$. The window function is usually chosen to be spherically symmetric

$g(k) = g_s(||k||_2)$ or separable $g(k) = g_x(k_x) \cdot g_y(k_y) \cdot g_z(k_z)$. The convolution (5.3) is to be evaluated at an evenly spaced set of locations $\{\tilde{k}_p\}$, while $f(k)$ is only known at the sample locations $\{k_m\}$. Hence, the convolution is evaluated as the summation:

$$(f * g)(\tilde{k}_p) = \sum_{m \in \mathcal{N}_W(\tilde{k}_p)} f_m \cdot g(\tilde{k}_p - k_m) \cdot \Delta k_m \tag{5.4}$$

where we define the neighbor sets $\mathcal{N}_W(\tilde{k}_p) \triangleq \{m : ||k_m - \tilde{k}_p||_2 < \frac{1}{2}W\}$ so that the summation is evaluated only over the kernel support window. The terms Δk_m are the discrete representation of the differential $d\kappa$ in (5.3), and are frequently called the density compensation factors (DCFs) in MRI applications. After computing the FFT of the equispaced samples $(f * g)(\tilde{k}_p)$, the desired equispaced samples of the approximate DFT of f are recovered by de-convolving by g. Via the Fourier-convolution identity, the de-convolution is computed as an inexpensive division by the Fourier transform of $g(k)$. This de-convolution step is also known as deapodization or roll-off correction.

The nuFFT achieves the log-linear arithmetic complexity of the FFT at the expense of accuracy. The re-sampling step can be understood as multiplication of the function $(f*g)(k)$ by a pulse-train function $\sum_j \delta(k - j \cdot \Delta\tilde{k})$, where $\Delta\tilde{k}$ is the spacing of the grid samples $\tilde{k}_p$ and $\delta(k)$ is the Dirac delta. Again via the Fourier-convolution identity, this is equivalent to a convolution in the Fourier domain with a pulse train with spacing $\frac{1}{\Delta\tilde{k}}$. This operation results in aliasing, as the n^{th} sample produced by the FFT is the sum $\sum_{j=1}^{\infty}(F \cdot G)(k_n - j \cdot \frac{1}{\Delta k})$, where G is the Fourier transform of the truncated resampling kernel $g(k)$. This aliasing is the source of NDFT approximation error. To reduce the effect of aliasing and increase NDFT approximation accuracy, the grid spacing must be finer than the Nyquist-rate of the signal f. If the FFT is computed at size $\tilde{N} > N$, then aliasing of the side-lobes of $F \cdot G$ achieves its maximum intensity in the margins rather than in the central N samples of the reconstructed signal. Thus the accuracy of the nuFFT is determined by a third parameter in addition to the two already mentioned ($g(\cdot)$ and W), usually defined as the grid oversampling ratio $\alpha = \tilde{N}/N$.

All three steps of the nuFFT (resampling, FFT, and de-convolution) are linear and can sensibly be written as matrix-vector products. Let $\mathbf{D}$ be an $N \times \tilde{N}$ diagonal matrix containing the deapodization weights, $\mathbb{F}$ be an $\tilde{N} \times \tilde{N}$ DFT matrix, and $\mathbf{\Gamma}$ be an $\tilde{N} \times M$ matrix containing the weights used during re-sampling for each grid/non-equispaced sample pair $\tilde{k}_p, k_m$:

$$\mathbf{\Gamma}_{p,m} = \begin{cases} g(\tilde{k}_p - k_m)\Delta k_m & ||\tilde{k}_p - k_m||_2 < \frac{W}{2} \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

With $\tilde{f} \in \mathbb{C}^M$ a vector of the non-uniform samples of the signal f and $y \in \mathbb{C}^N$ a vector of the pixels in the approximated DFT of f, the nuFFT is equivalent to computing the matrix-vector product:

$$y = \mathbf{D}\,\mathbb{F}\,\mathbf{\Gamma}\,\tilde{f} \tag{5.6}$$

The adjoint nuFFT is equivalent to multiplication by the conjugate transpose, $\mathbf{\Gamma}' \mathbb{F}^* \mathbf{D}$.

The deapodization is very inexpensive, consisting of a single multiplication per grid point. The FFT is expensive, but highly efficient library implementations are widely available. Thus much of the difficulty in the efficient implementation and performance optimization of the nuFFT lies in the resampling step, or equivalently of the matrix-vector product $\mathbf{\Gamma}\, \tilde{f}$.

5.2.1 Algorithmic Complexity

The following sections will discuss the implementation and performance optimization of the nuFFT. To motivate the discussion, it is useful to discuss the asymptotic runtime of the algorithm. We assume that the input is a non-equispaced sampling of f and that the desired result is an equispaced sampling, so that the resampling must only be performed once. The number of arithmetic operations to be performed depends on the input size M and output size N, but also on the parameters that control NDFT approximation accuracy: the kernel function $g(\cdot)$, the convolution truncation width W, and the grid oversampling ratio α (equivalently, the oversampled grid size $\tilde{N}$).

With d-dimensional coordinates (typically 2-D or 3-D), the resampling convolution performs approximately W^d kernel evaluations and multiplications per input sample, so the number of arithmetic operations performed by the the nuFFT is

$$C_R M W^d + C_F \tilde{N} \log \tilde{N} + N \tag{5.7}$$

where C_R and C_F are constant factors associated with the implementation of resampling and the FFT, respectively. The resampling constant C_R depends on the method by which the implementation evaluates $g(\cdot)$, which may require evaluation of transcendental or trigonometric functions. C_R also depends on the dimensionality d, as the formula for the size of the convolution window differs for 2D and 3D. C_R also depends on the amount of precomputation the implementation performs as described in Section 5.5.

The linear deapodization cost – the third term in (5.7) – is negligible, and the majority of the nuFFT's runtime is spent in performing the resampling and FFT. However, it is unclear from Equation (5.7) which of these two operations will dominate the nuFFT's runtime. A well-known property of the nuFFT is that an implementation can effectively trade runtime between the two, and force either to dominate. To achieve a given level of accuracy, one can choose to make the window width W very large. This tends to decrease the amplitude of the sidelobes of G, and consequently a smaller grid size $\tilde{N}$ is tolerable. Alternately, a small window W produces large sidelobes and necessitates a larger grid. Section 5.3 describes our strategy for identifying the tradeoff between W and α that minimizes the total runtime. This strategy will also allow us to decide several other implementation options for which the asymptotic analysis provides no guidance.

5.3 Empirical Performance Optimization

Section 5.2.1 discusses the inherent runtime tradeoff between the resampling convolution and the Fourier transform. However, the asymptotic analysis is unable to suggest a strategy for selecting the parameters that guarantee the best performance. Many possible values of the nuFFT's parameters are able to achieve any given level of NDFT approximation accuracy, but this space of error-equivalent implementations will exhibit very different balances of runtime between resampling and the FFT. The actual runtime an implementation achieves is a complicated function not only of the nuFFT parameters, but also of the microarchitecture of the target platform and the non-uniform sampling pattern, which determines the pattern of memory access during resampling. The execution pipelines and memory systems of modern processors are highly optimized for particular types of instructions and patterns of memory access. Cache behavior in particular is very sensitive to the order in which data is accessed by a program, and cache performance is notoriously difficult to model or predict.

We propose empirical performance measurement as a strategy to resolve the ambiguity in nuFFT parameter selection. This approach has been called *auto-tuning* in other contexts, as it has been successfully applied to a number of other important numerical algorithms [85, 83, 21, 20, 33, 68]. In all these cases, one can enumerate a number of functionally-equivalent implementations of an algorithm but cannot *a priori* determine which will provide optimal performance. Auto-tuning approaches benchmark a number of the alternate implementations in order to make performance optimization decisions. This benchmarking phase is commonly called *planning*, a term first used by the well-known and widely used FFTW library [33]. In cases where the performance of the algorithm is input-dependent, planning may need to be performed on-line. In the case of the nuFFT, auto-tuning must be performed for each pattern of non-uniform sampling. However, the resulting auto-tuned implementation can be reused for multiple signals sampled at the same set of locations.

We design the planning phase for the Gridding nuFFT to take into account the size of the final signal N, the non-equispaced sample locations $\{k_m\}_{m=1}^{M}$, and the desired level of NDFT accuracy measured via the metric discussed in Section 5.4.1. Section 5.6 discusses the motivation of this decision by the application of the nuFFT in MRI reconstruction. However, the FFT and Sparse Matrix libraries on which we build our implementation also have planning phases, thus it is natural that we require a planner as well. Our functionally equivalent implementations differ in their choices of the resampling kernel $g(\cdot)$, the convolution window width W, and the size of the oversampled grid $\tilde{N}$. Additionally, Section 5.5 will discuss several approaches of implementing the resampling convolution on parallel systems. The optimality of one approach over the others depends on characteristics of the microarchitecture: cost of interthread synchronization, effective memory bandwidth, and floating-point execution rates.

Much of the efficiency of nuFFT algorithms is due to the use of the FFT for computing the Fourier transform. Similarly, Section 5.5 discusses the resampling can be implemented very efficiently as a matrix-vector multiplication. High-performance auto-tuning libraries are

available for computing both the FFTs and matrix-vector products. The nuFFT planner for the nuFFT must invoke the planner for these libraries as well. The FFTW library [33] produces highly efficient FFTs by searching over possible Cooley-Tukey factorizations of the transform size. To avoid an exhaustive search over this space of implmenetations, FFTW relies on a very effective dynamic programming heuristic that avoids benchmarking a particular problem size more than once. When considering all possible recursive Cooley-Tukey factorizations of a problem size, the FFTW planner will encounter smaller transform sizes many times. Each time the planner encounters a particular transform, it will be performed at possibly different input and output strides. Although memory-system behavior depends on these strides, the dynamic programming heuristic provides close to optimal performance.

Similarly, the Optimized Sparse Kernel Interface (OSKI) [83] library provides a heuristically auto-tuned implementation of sparse matrix operations such as the matrix-vector multiplication that can be used to implement resampling in the nuFFT. Many of the optimizations that OSKI can perform involve modifying the sparse matrix data structure to achieve higher memory system performance during matrix products. For example, general-purpose matrix data structures such as Compressed Sparse Row (CSR) generally store at least one integer index per nonzero entry in the matrix. However, many matrices exhibit patterns of local density, and indexing overhead can be reduced by storing a block of nonzeros in a small dense matrix. Rather than storing a single integer per nonzero, the sparse matrix data structure can store an integer per dense block. Using this more parsimonious data structure reduces memory traffic and increases the length of unit-stride memory accesses during matrix-vector products, potentially improving performance.

5.4 Parameter Selection

Selection of the three nuFFT algorithmic parameters – interpolation kernel $g(\cdot)$, convolution width W, and grid oversampling factor α – determines the balance between NDFT approximation accuracy and runtime, and the balance of complexity between resampling and the FFT. Per Equation (5.7), the asymptotic complexity depends on W and α, but only constant factors are affected by the choice of $g(\cdot)$. Moreover most implementations pre-sample the kernel function and estimate its value during resampling by linear interpolation, so the choice of $g(\cdot)$ is of little import for the purposes of performance optimization. Consequently, for the remainder of this work we'll assume that $g(\cdot)$ is the Kaiser-Bessel window:

$$g(k) \quad = \quad \frac{\tilde{N}}{W} I_0(\beta\sqrt{1-(2\tilde{N}k/W)^2}) \tag{5.8}$$

Where β is a scalar parameter, $I_0(\cdot)$ is the zero-order modified Bessel function of the first kind, and $\tilde{N}$ is the oversampled grid width. The inverse Fourier transform of Equation (5.8)

to be used during deapodization is computed as:

$$G(x) \;=\; \frac{\sin\sqrt{(\pi W x/\tilde{N})^2-\beta^2}}{\sqrt{(\pi W x/\tilde{N})^2-\beta^2}}.$$

Previous works [44, 7] have determined this gridding function (with appropriate selection of β) to provide near optimal image quality results. In this work, we use the β from Beatty *et al.* [7]: $\beta_{\alpha,W} = \pi\sqrt{\frac{W^2}{\alpha^2}\left(\alpha-\frac{1}{2}\right)^2-0.8}$. This expression is derived by positioning the first zero-crossing of $G(x)$ at $\tilde{N}-N/2$ as a heuristic to control aliasing behavior.

5.4.1 Aliasing Amplitude

Our auto-tuning approach requires that the desired NDFT approximation accuracy be specified to the planner, so that we can enumerate a set of values for α and W that explore the error-equivalent complexity tradeoff between resampling and the FFT. The most natural metric by which to measure NDFT approximation accuracy is the difference between the output of the nuFFT and the direct computation of the NDFT via (5.1). However, this metric is expensive to compute and dependent on the signal to be reconstructed. It is highly desirable to evaluate image reconstruction quality as a function only of the selected algorithmic parameters $g(\cdot)$, α, and W, especially since our planning approach does not require the sample values to be known. Also we desire a metric with a closed-form analytic expression which we can more easily use to constrain the space of parameters our planner must evaluate.

Beatty *et al.* [7] proposed the *aliasing amplitude* metric, which achieves these desired properties. Rather than measuring the reconstruction error for any particular signal, the aliasing amplitude metric estimates the amount of energy this aliasing produces at each pixel in the image:

$$\varepsilon(x_i) \;=\; \sqrt{\frac{1}{G(x_i)^2}\sum_{p\neq 0}\left[G(x_i+mp)\right]^2} \tag{5.9}$$

In the MRI application that motivates this work, our primary accuracy-concern is the absence of reconstruction artifacts. Thus the maximum aliasing amplitude $\varepsilon^* = \max_{x_i}\varepsilon(x_i)$ is a fitting metric. Figure 5.1 demonstrates the relationship between this quality metric and the parameters W and α, using the Kaiser-Bessel kernel. Given a desired accuracy, fixing either one of W of α specifies a bound on the value of the other. *E.g.* one can read from Figure 5.1 the minimum kernel width necessary to achieve a given error tolerance with a given oversampling ratio. For the purposes of planning an nuFFT implementation, the aliasing amplitude metric allows us to easily enumerate an arbitrarily large set of error-equivalent choices for W and α. The planner can evaluate as many (W,α) pairs as are feasible in the time allotted for planning.

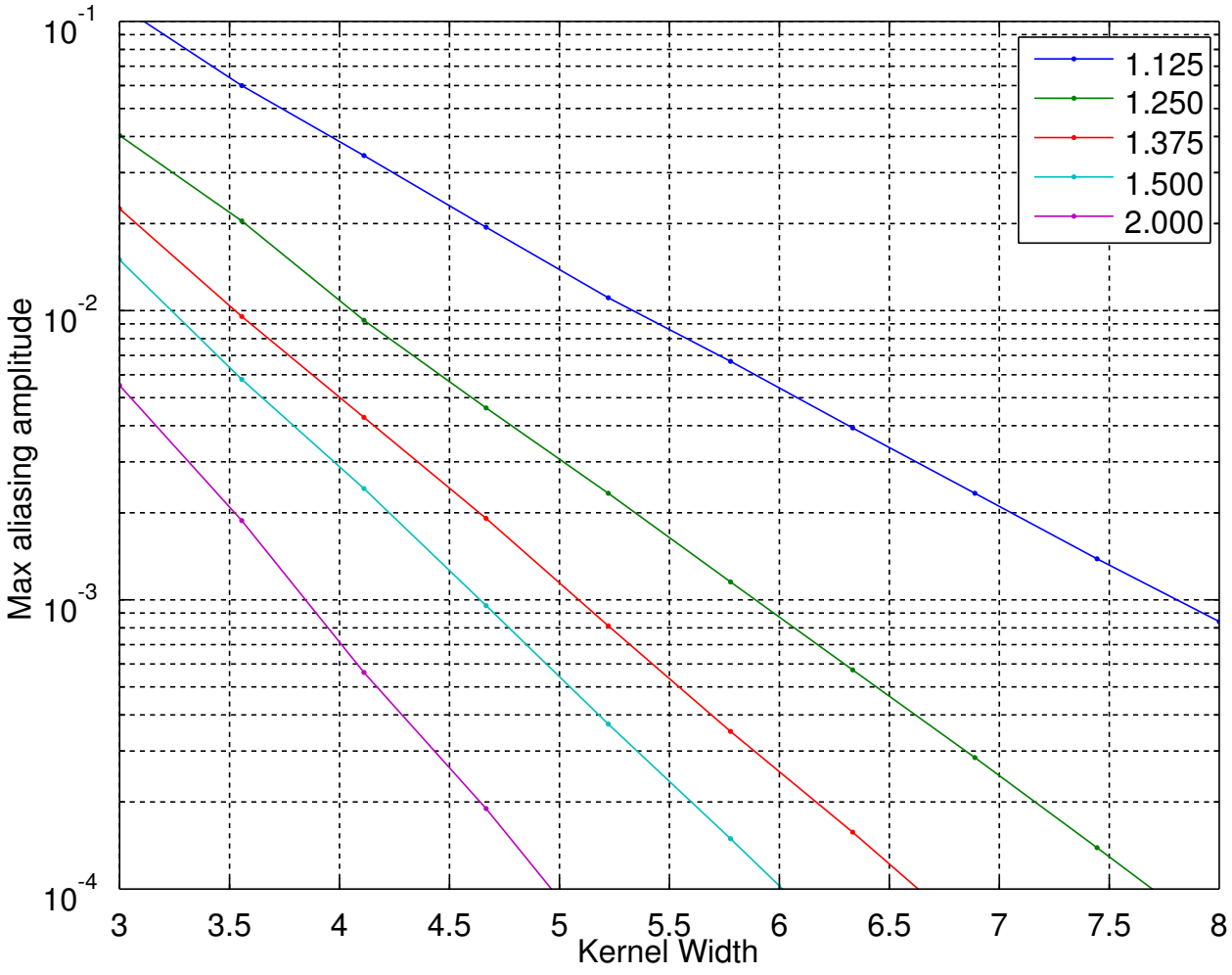

Figure 5.1: Plot of max aliasing amplitude vs. kernel width for a variety of α. Plots are generated by evaluating equation (5.9) in 1 spatial dimension for a variety of values of α and W. The infinite sum is truncated to $|p| \leq 4$.

$\{f_m, k_m\}$: Non-equispaced samples of $f(x)$
Δk_m : Density Compensation Factors
$g(k)$: Resampling kernel
$\{h_p, \tilde{k}_p\}$: Equispaced samples of $(f * g)(k)$

Evaluation at equispaced locations
(1) for $m = 1, \ldots, M$
(2) for $p \in \mathcal{N}_W(k_m)$
(3) $h_p \leftarrow h_p + f_m \cdot g(\tilde{k}_p - k_m) \cdot \Delta k_m$

Evaluation at non-equispaced locations
(4) for $m = 1, \ldots, M$
(5) for $p \in \mathcal{N}_W(k_m)$
(6) $f_m \leftarrow f_m + h_p \cdot g(\tilde{k}_p - k_m) \cdot \Delta k_m$

Figure 5.2: Pseudocode of the precomputation-free resampling, which implements the matrix-vector multiplication $h \leftarrow \mathbf{\Gamma} f$ without storage besides the source data f and the output grid h. The sets $\mathcal{N}_W(k_m)$ are the indices of equispaced samples within radius $\frac{1}{2}W$ of the sample location k_m, and are enumerated by iterating over the volume whose bounding box is $(\lfloor k_m - \frac{1}{2}W \rfloor, \lceil k_m + \frac{1}{2}W \rceil)$. Shared-memory parallel implementations distribute iterations of the m-loop on line (1) among threads. Since the set of grid locations updated in line (3) by a given non-equispaced sample are not known *a priori*, parallel implementation must protect the grid updates on line (3) via hardware-supported mutual exclusion. Note that the update in line (6) does not require mutual exclusion.

5.5 Resampling Implementation

In this section we discuss the implementation of the convolution summation in Equation (5.4), or equivalently the multiplication by the matrix $\mathbf{\Gamma}$ defined in Equation (5.5). We'll review several previously proposed approaches from the MRI literature, and describe them via their relation (explicit or implicit) to the matrix $\mathbf{\Gamma}$. We organize this discussion according to the amount of precomputation performed by each approach. All precomputation approaches perform off-line analysis of the non-equispaced sample locations, and can be performed before the actual sample values are available. If the nuFFT is to be performed for multiple signals that have been sampled at the same set of locations, then the precomputation need only be performed once. The cost of the precomputation is amortized, and its performance benefit multiplied, over the multiple nuFFT executions.

5.5.1 Precomputation-free Resampling

One can directly translate Equation (5.4) into an algorithm for performing the resampling as a convolution, and pseudocode of this approach appears in Figure 5.2. The primary benefit of this approach is its lack of additional storage beyond the input and output data. To reduce runtime in many practical applications, the interpolation kernel $g(k)$ is pre-sampled in the range $-\frac{1}{2}W < ||k||_2 < \frac{1}{2}W$ and the values stored in a table. The size of this table is negligible compared to the samples of f. During resampling, values of $g(k)$ are estimated via linear interpolation between adjacent table entries. This linear interpolation requires only a few arithmetic operations, whereas many commonly used interpolation kernels are transcendental and relatively expensive to evaluate. Shared-memory parallel implementations of this approach partition the non-equispaced samples among processing threads. If the resampling is to be evaluated at the non-equispaced locations, then this parallelization requires no synchronization among threads. However, when resampling onto the equispaced grid it is not known *a priori* which grid locations h_p will be updated by a given sample f_m. If two threads are concurrently computing grid updates for sample locations whose distance is less than the kernel width W, then their grid updates will overlap. The memory systems of modern parallel processors cannot guarantee correct behavior in this case unless the grid updates are protected by mutual exclusion. The grid updates are very frequent, and primitives such as spin-locks and semaphores incur much too high a performance penalty to be used. Instead, low-level processor-specific read-modify-write operations must be used. Several prior works have described this implementation on GPGPU systems, and it is equally applicable in multi-core CPU systems. In the case of a Cuda [62] implementation, the `AtomicAdd()` library function enables efficient parallelization. For x86 CPU implementations, equivalent functionality is provided by the `cmpxchg` instruction with a `LOCK` prefix inside of an execute-retry loop. Most compilers provide intrinsic functions to simplify use of these instructions.

5.5.2 Fully-Precomputed Resampling

Another implementation option provided by several previous nuFFT libraries [31, 46] is to perform the re-sampling as a multiplication of the $\mathbf{\Gamma}$ matrix defined in Equation (5.5) by the vector containing the sample values f_m. The values of the entries of $\mathbf{\Gamma}$ depend only on the locations $\{k_m\}$, of the non-equispaced samples, the density compensation factors $\{\Delta k_m\}$, and the nuFFT algorithmic parameters $g(\cdot)$, W, and α as defined in Section 5.2. Computing and storing all MN entries of $\mathbf{\Gamma}$ is unnecessary, since the kernel width W is typically much smaller than the size of the equispaced grid and the majority of entries $\mathbf{\Gamma}_{p,m}$ are zero – i.e. $\mathbf{\Gamma}$ is a sparse matrix. Typically in MRI applications W is chosen to be 2-10 times the spacing of grid samples, whereas the grid is 200-500 samples wide. Sparse matrices are used widely in numerical computing applications, and can be stored parsimoniously in data structures that permit efficient, synchronization-free implementation of matrix-vector products.

The potential benefit of the precomputed-matrix implementation over the precomputation-

free implementation is two-fold. First, the widespread use of sparse matrices in scientific applications has produced a number of very efficient libraries of matrix-vector operations. While a detailed discussion of the performance optimization of sparse matrix operations is out of the scope of this work, several pertinent features of these libraries deserve our attention. These libraries typically support several storage formats, such as the general-purpose Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) that we use in our implementation. Additionally some libraries include storage formats optimized for particular patterns of nonzero locations, for example explicitly storing some zeros in order to store submatrices densely and reduce indexing overhead. Just as we rely on highly optimized extant implementations of the FFT algorithm, we can rely on highly optimized implementations of matrix-vector products. The existence of these libraries greatly simplifies implementation of the nuFFT, as all low-level performance optimization issues are managed by the FFT and sparse matrix libraries.

Second, sparse matrix-vector multiplication (SpMV) performs substantially fewer arithmetic operations than does the convolution-based implementation of the resampling described in Figure 5.2. Each nonzero entry $\mathbf{\Gamma}_{p,m}$ corresponds to a pair of samples, (f_m, k_m) from the non-equispaced data and $(h_p, \tilde{k}_p)$ from the equispaced grid, whose distance is less than the resampling kernel radius. A sparse matrix-vector multiplication performs a single multiplication and addition per nonzero entry in the matrix. On the other hand, the precomputation-free implementation must perform that same multiply-accumulate after enumerating the neighbor sets $\mathcal{N}_W(\cdot)$, evaluating the interpolation kernel $g(\cdot)$, and multiplying by the density compensation Δk_m. The number of arithmetic operations required depends on the implementation of $g(\cdot)$, but the commonly used approach is to linearly interpolate samples of a spherically symmetric function $g_s(\cdot)$. This approach requires 15 arithmetic evaluations (9 in computing the distance $||k_m - \tilde{k}_p||_2$ and 6 in the linear interpolation) as well as evaluation of the transcendental square root. An implementation that uses a separable interpolation kernel, rather than a spherically symmetric kernel, would perform slightly fewer arithmetic operations during convolution, but would still benefit from precomputation.

While precomputation of $\mathbf{\Gamma}$ decreases arithmetic operation count, the matrix datatsructure has a large memory footprint. Since each nonzero entry in this data structure must be accessed during resampling, the matrix-vector product incurs a large volume of memory traffic. As has been noted by recent literature on the optimization of sparse matrix operations [85], this memory traffic is the performance-limiting factor of SpMV on modern shared-memory parallel processors. Thus, the performance benefit of precomputation is contingent upon the ability of the processor's memory system to satisfy the memory traffic of SpMV faster than its floating-point pipelines can perform the convolution.

5.5.3 Partially Precomputed Resampling

Several previous works have presented approaches that perform some pre-analysis in order to more efficiently compute the resampling, but do not precompute the entire matrix $\mathbf{\Gamma}$. We do

not include these approaches in the performance evaluation in Section 5.7, however the empirical search strategy we describe in Section 5.3 can easily accommodate them. We describe two relevant GPGPU implementations that have attempted to obviate the inter-thread synchronization needed by the precomputation-free implementation described in Section 5.5.1. A simple way to achieve this goal is to precompute only the locations of the nonzeros in $\mathbf{\Gamma}$, i.e. the pairs (p, m) for which $\mathbf{\Gamma}_{p,m} \neq 0$. The implementation of resampling would need to re-evaluate the values of the nonzeros. While this approach disambiguates grid-update conflicts, it does not achieve any performance gain: the size of the data structure required is almost as large as $\mathbf{\Gamma}$, and this approach still incurs the runtime cost of evaluating the non-zeros.

The approach presented by Sørensen *et al.* [73] achieves a synchronization-free implementation of the convolution in Figure 5.2 by precomputing the locations of all the nonzeros in $\mathbf{\Gamma}$, but reducing the size of the data structure with two crucial transformations. First, they partition the grid samples $\tilde{k}_p$ (equivalently, partition the rows of $\mathbf{\Gamma}$) among the processing threads. By taking the union of column indices m in a partition, they find non-disjoint subsets of the non-equispaced samples from which each thread interpolates onto the grid. Second, they sort and run-length-encode the resulting column-index sets to further reduce their memory footprint. This approach is a highly attractive alternative to the precomputation-free convolution on systems that lack the highly efficient read-modify-write instructions mentioned in Section 5.5.1. This was the case for the early GPGPU system targeted by Sørensen *et al.*.

The "spatial binning" approach of Obeid *et al.* [64] is inspired by algorithms for molecular dynamics problems that compute with spatially-local interactions among points at arbitrary locations in $\mathbb{R}^d$. In MRI applications, the samples are usually stored in readout-order. Obeid *et al.* propose to partition the grid samples into contiguous rectangular regions (bins), and to permute the non-equispaced samples so that all samples within a given bin are stored contiguously. From this re-ordering, which is equivalent to a radix sort, one can infer the interacting grid/sample pairs (i.e. locations of nonzeros in $\mathbf{\Gamma}$).

The approach of Fessler *et al.* [31] achieves some of the arithmetic-operation reduction provided by the sparse matrix precomputation, rather than attempting to reduce inter-thread synchronization costs. Fessler *et al.* describe a particular choice of the resampling kernel $g(\cdot)$ that can be partially precomputed in one of several options, reducing the number of arithmetic evaluations that must be performed during resampling. Some of these options impact NDFT approximation accuracy.

5.6 Planning and Precomputation in MRI Reconstruction

Our work is primarily motivated by Gridding reconstruction in MRI, where the nuFFT is used to compute the adjoint NDFT defined in Equation (5.2). Our auto-tuning approach is

highly applicable in this context, as the information required by the planner is known far in advance of the values of the samples. Additionally, it may be possible to perform much of the tuning during installation of the reconstruction system. For example, the auto-tuning of the equispaced FFT can be performed independently of the auto-tuning of the sparse matrices. The FFT can potentially be auto-tuned for all possible image matrix sizes to be used during clinical imaging, and the resulting plans saved in "wisdom" files [33]. In many clinical applications it may also be possible to enumerate the set of non-Cartesian trajectories that will be used, since the meaningful FOVs and resolutions are limited by patient anatomy. Sparse matrices can be precomputed for these trajectories and also stored in the file system.

If a pre-existing plan for a trajectory is not available, it may be possible to entirely hide the cost of planning and precomputation by performing it simultaneously with data acquisition. During an MRI scan, samples F_n are acquired in the Fourier domain (k-space) at a set of locations $\{k_n\}$ defined by the Gradient pulse sequence. The locations $\{k_n\}$ are known when the scan is prescribed, but the sample values F_n are not known until after the scan completes. MRI data acquisition is inherently slow, and the interval between prescription and completion allows ample time for precomputation. For example, data acquisition in high-resolution, physiologically gated/triggered 3D imaging can take several tens of minutes. Even in real-time 2D imaging applications, where the acquisition of each signal to be reconstructed can take only a few tens of milliseconds, the same trajectory is used for continuous data acquisition over a much longer period of time.

Any performance benefit provided by planning and precomputation will be multiplied by the number of nuFFT executions to be performed during reconstruction. Most MRI applications use Parallel Imaging, where the k-space samples are acquired simultaneously from multiple receiver channels. Each channel acquires the same set of k-space sample locations, but the signals are modulated by the spatial sensitivity profiles of the receiver coils. The nuFFT must be computed for each channel individually. Additionally, many recently proposed MRI reconstructions use iterative algorithms to solve linear systems that model the data acquisition process. These algorithms typically must compute two nuFFTs (a forward and an adjoint) per iteration for each Parallel Imaging channel. State-of-the-art parallel imaging systems utilize up to 32 channels, and iterative reconstructions can perform 10-100 iterations. Thus even moderate performance improvement can substantially decrease reconstruction runtimes.

5.7 Performance Results

We demonstrate the effectiveness of our implementation strategy via a number of sampling trajectories representative of non-Cartesian MRI reconstruction workloads. Figures 5.3, 5.4, and 5.5 describe in detail the results of performance tuning for three types of MRI sampling trajectories, and Figures 5.6, 5.7, and 5.8 describe the performance achieved for a wider range of trajectories.

All performance results are collected on our test system, which has dual-socket × six-core Intel Westmere CPUs at 2.67 GHz with 64 GB of system DRAM. GPU performance results are collected on an Nvidia GTX580 GPU with 3 GB of high-speed Graphics DRAM. We are not evaluating hybrid CPU-GPU implementations, and reported runtimes include no PCI-Express transfer overhead. Sparse matrices are represented in Compressed Sparse Row (CSR) format, and matrix-vector multiplications are performed via optimized libraries. Our CPU implementation uses the Optimized Sparse Kernel Interface version 1.01h (OSKI) [83], and our GPU implementation uses the CUSparse library [9] distributed with Cuda version 4.0. OSKI only provides sequential implementations, and we parallelize the matrix-vector multiplication by partitioning the rows of $\mathbf{\Gamma}$ among the 12 CPU cores. For load balance, we choose the partition to approximately equalize the number of non-zeros in each CPU's set. We use the FFTW 3.0 library [33] for computing FFTs on the CPUs, and tune with the `FFTW_MEASURE` flag. The sequential baseline against which we report speedups also uses tunes FFTs with `FFTW_MEASURE`. We use the CUFFT [62] library for the GPU implementation.

Performance Tuning Experiments Figure 5.3 shows nuFFT performance for a 3D cones [38] trajectory, and has M = 14,123,432 sample points and a maximum readout length of 512. Figure 5.4 shows performance for a very large radial trajectory [10] with M = 26,379,904 sample points. Figure 5.5 shows results for a 2D spiral [56] trajectory with M = 38,656 sample points. All three trajectories are designed for 25.6 cm field of view with 1 mm resolution, isotropic, and the final image matrix size N is 256^d. In all three cases, we set the maximum aliasing amplitude to be 1e-2. These figures show four panels each. The top-left panels show performance achieved on the CPUs in our test system. The top-right panels show performance achieved on a GPGPU in the test system.

Both of these panels plot the runtime required by two implementations of the resampling (as a convolution and precomputed sparse-matrix), and the time required for the FFT. These runtimes are plotted versus a range of values for the grid oversampling ratio α between 1.2 and 2.0, and total nuFFT runtime is the sum of resampling runtime, FFT runtime, and the negligible deapodization runtime. The bottom-left panel shows the memory footprint required by both the sparse matrix and the oversampled grid, also plotted versus α. We use the same CSR matrix data structure in both our CPU and GPU implementations, so the memory footprint is the same in both cases. The bottom-right panel compares the best performance achieved by the four types of implementations (convolution/sparse-matrix on CPU/GPU) to a baseline: a single-threaded convolution-based CPU implementation with the value of α that maximizes performance.

Figure 5.3 demonstrates several trends that correlate well with our expectations. Smaller values of α result in small grid sizes, but large sparse matrices and long convolution runtimes. Conversely, large values of α increase grid size but decrease resampling runtime. The CPU-parallelized convolution implementation consistently achieves approximately 10× performance improvement over the sequential baseline, while the sparse-matrix precomputation

provides a substantial 5× further speedup. Interestingly, the precomputed sparse matrix provides performance comparable to the GPU-based convolution, and matrix precomputation provides a 2× speedup on the GPU. That the precomputation provides a relatively smaller benefit on the GPU verifies our claim that the relative memory bandwidth and instruction throughputs of a processor determine the usefulness of this optimization. GPUs have substantially higher arithmetic throughput than CPUs (relative to the memory bandwidths of the two systems), and thus trading memory traffic for floating-point computation is less beneficial on GPUs. FFT performance is highly sensitive to grid size. Most importantly, it is not monotonic: FFT libraries perform better when the grid size factors into small prime numbers. The anomalously bad performance of the GPU FFT in the second- and third- highest values of α correspond to matrix sizes of 491 (prime) and 502 (almost prime – 2×251).

The same trends are visible in Figure 5.4. However the Radial trajectory's sampling density is much higher in the center of k-space, and consequently the trajectory has almost twice as many samples as the Cones trajectory. The memory footprint of the sparse matrix is correspondingly larger than that of the Cones trajectory, and overflows the GPU's 3GB memory capacity for all values of α that we evaluated. Since the CSR data structure requires an integer to be stored for each grid point, the size of the sparse matrix begins to increase as α grows further. Hence, the sparse-matrix implementation of this radial trajectory is infeasible for our GPUs. Note that even in Figure 5.3 the sparse matrix implementation is infeasible for small values of α. However, the much larger memory capacity of the CPUs in the system permits the fully-precomputed implementation in most cases. Thus CPU performance for this trajectory is about 15% better than the GPU performance.

Performance for a Range of Trajectories In Figures 5.6, 5.7, and 5.8, we have benchmarked the performance achieved by our nuFFT implementation for a number of 3D Cones trajectories with isotropic FOV varying from 16 cm to 32 cm. These figures show performance results for tuning on the CPUs of the system described above, using all 12 available processor cores. All these trajectories are designed for 1 mm isotropic resolution. Thus, as the FOV increases, the sizes of the trajectory and final image matrix also increase. For each FOV, we evaluate oversampling ratios in the range of 1.2 to 2.0. Just as above, all nuFFTs are computed at 1e-2 maximum aliasing amplitude. Figure 5.6 plots three data series describing the performance-optimal precomputation-free implementations chosen by our approach. The blue points display the runtime of the FFT, the red points display the runtime of the resampling convolution, and the magenta points display the total runtime of the nuFFT. Figure 5.7 plots the runtimes achieved using a precomputed Sparse Matrix to perform the resampling. Similarly, the blue data series represents the runtime of the FFT for the chosen image matrix size, the green series shows the runtime of the corresponding matrix-vector multiplication, and the magenta series shows the total nuFFT runtime. Figure 5.8 compares the oversampling ratios that achieve optimal performance for the convolution-

based and sparse matrix-based nuFFTs shown in Figures 5.6 and 5.7. The red points show the oversampling ratios chosen for the convolution-based nuFFT, and the green points show the ratios chosen for the sparse matrix nuFFT.

From Figure 5.6 it is clear that in a precomputation-free nuFFT implementation, the runtime of the convolution dominates that of the FFT. The performance-optimal oversampling ratio is very high, as shown in Figure 5.8, because it is beneficial to increase the image matrix size and FFT runtime while decreasing the convolution runtime. Still, the convolution runtime grows much more rapidly than the FFT runtime, and runtimes can be greatly improved by precomputing the sparse matrix. Figure 5.7 shows that all of the trajectories examined can reap the benefit of sparse matrix precomputation. Figure 5.8 shows that the tuning phase consistently selects a much smaller grid oversampling ratio for the sparse matrix implementation. Consequently, the runtime is much more evenly balanced between the FFT and the matrix-vector multiplication. Consistent with the results shown above in Figures Figures 5.3, 5.4, and 5.5, a precomputed sparse matrix enables substantially faster implementation of the nuFFT's resampling.

Heuristic Tuning In some applications, the time spent in exhaustive evaluation of a wide range of grid oversampling ratios is prohibitive. Note that evaluating each oversampling ratio requires the computation of the sparse matrix, which requires computing, storing, and sorting all nonzeros by row-index to convert to CSR format. Computing many such matrices can be time-consuming, and it is desirable to have simple heuristics to prune the search space. Since precomputing the sparse matrix shifts much of the computational burden onto the FFT, we propose a simple heuristic that selects the oversampling ratio that minimizes the FFT runtime. This heuristic does not take into consideration the runtime of the corresponding matrix-vector multiplication. Since the FFT can be benchmarked during system installation, this heuristic requires no computation during planning except for precomputation of a single sparse matrix. The blue data series in Figure 5.8 plots the oversampling ratios chosen by this heuristic, and Figure 5.10 compares the nuFFT runtimes resulting from this heuristic to those resulting from exhaustive evaluation of a range of grid oversampling ratios. In many cases, the heuristic chooses the minimum ratio evaluated – in our case, 1.2. However, due to the pathologically poor performance of the FFT for some transform sizes, this heuristic occasionally chooses a larger grid size. In some cases, the heuristic chooses the same grid size as the exhaustive search strategy. As shown in Figure 5.10, performance is comparable between the two approaches.

Memory-Limited Heuristic Some applications may wish to limit the memory footprint of the Gridding algorithm. In this case, one can use a modified heuristic that chooses the grid oversampling ratio to minimize FFT runtime subject to a specified limit on matrix size. This will force a larger grid oversampling ratio to be used and degrade performance relative for larger trajectories, as demonstrated by the cyan series in Figure 5.10. Note that the size

of the grid and sparse matrix can both be computed in $O(1)$ time, and this heuristic incurs very little cost during planning. However, we note that even for the largest trajectories we evaluate here, this memory-limited heuristic can achieve superior performance to the precomputation-free nuFFT when 2 GB of memory usage is allowed. In applications where memory is more severely restricted, sparse matrix precomputation may be infeasible. In these cases, a memory-limited heuristic can still be used to provide acceptable runtimes. In Figure 5.11, we demonstrate the runtime achieved for our range of 3D Cones trajectories when matrix memory footprint is limited to 1 gigabyte. For all trajectories with FOV 24 cm or larger, none of the evaluated grid oversampling ratios produces a sparse matrix that satisfies this severe memory limit, and the nuFFT must rely on a convolution-based implementation of resampling. Figure 5.11 evaluates a heuristic that chooses oversampling ratio in the range 1.5–2.0 to minimize FFT runtime when sparse matrix precomputation is infeasible.

5.8 Discussion and Conclusion

We have discussed the Gridding non-uniform Fourier transform approximation used in MRI reconstruction of non-Cartesian data, and described the runtime tradeoff between resampling and the FFT. This tradeoff, combined with the several methods of resampling implementation, motivates an *auto-tuning* approach to performance optimization. Following similar approaches used by other numerical libraries, we perform optimization in a *planner* routine that requires only the specification of the non-uniform sampling trajectory and desired level of Fourier transform approximation accuracy. The resulting optimized implementation can be re-used for the Fourier transforms of many signals sampled at the same set of locations. On modern processors, highest performance is always acheived by precomputing a sparse matrix whose matrix-vector multiplication performs the resampling. In this case, most of the arithmetic operations performed during resampling can be computed during the planner routine, although the data structure storing the matrix has a sizeable memory footprint. However, performing the nuFFT via a precomputed sparse-matrix is over 50× faster than an optimized, sequential baseline. A GPU implementation can be over 100× faster than baseline if the GPU's memory capacity permits storage of the sparse matrix. To guarantee optimal performance, matrices corresponding to a range of oversampling ratios must be evaluated. In some applications, it may be undesirable to perform this exhaustive evaluation. We have presented an inexpensive heuristic that achieves near-optimal performance and requires evaluation of only a single matrix.

A number of other implementation strategies and optimizations can readily be incorporated into the auto-tuning strategy. For example, we did not evaluate the performance of the partial-precomputation approaches discussed in Section 5.5.3. Moreover the implementation of resampling as a sparse matrix permits other optimizations as well, for example batching of multiple resampling instances into a single sparse-matrix/dense-matrix multiplication. Such strategies may improve performance beyond what we report here, and we leave their evalua-

tion to future work. The high degree of flexibility afforded by the empirical-optimization, or auto-tuning, approach additionally future-proofs our library: empirical search will be able to account for microarchitectural changes in future processor generations that radically change the implementation parameters that achieve optimal performance. Thus, we can sensibly call our implementation the *fastest* nuFFT.

5.9 Acknowledgments

Many thanks to David Sheffield for providing the CPU implementation of the atomic grid update operation, and to Holden Wu for providing the 3D Cones trajectory design software.

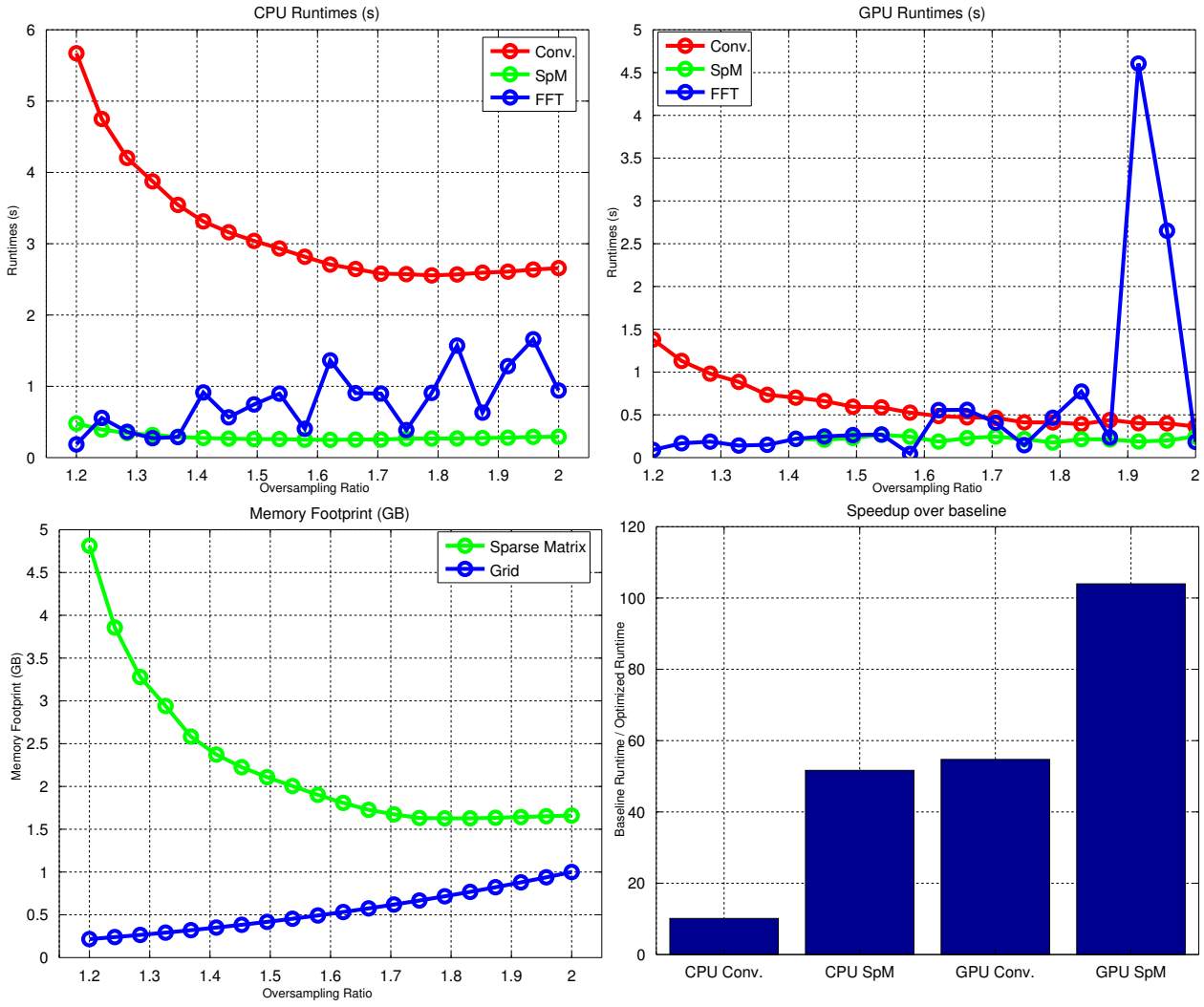

Figure 5.3: nuFFT auto-tuning performance for the 14M-sample 3D Cones trajectory. The top-left and top-right panels show runtimes vs. grid oversampling ratio α separately for convolution-based resampling (red), matrix-vector resampling (green), and FFT (blue). The top-left panel shows multi-core CPU runtime, while the top-right panel shows GPU runtime. The bottom-left panel shows the memory footprint of the oversamped grid (glue) and the sparse matrix (green). Memory footprint is identical for CPU and GPU systems. The bottom-right panel shows nuFFT speedup over optimized baseline of four implementations: on CPU with convolution-based resampling, on CPU with matrix-vector resampling, on GPU with convolution, and on GPU with matrix-vector. All four implementations and baseline use a performance-optimal oversampling ratio. For the GPU implementation, note that oversampling ratios below 1.4 produce matrices larger than the memory capacity of the GPU used in our evaluation. Also note that the CPU implementation with matrix-vector resampling achieves performance nearly equal to that of the GPU with convolution-based resampling.

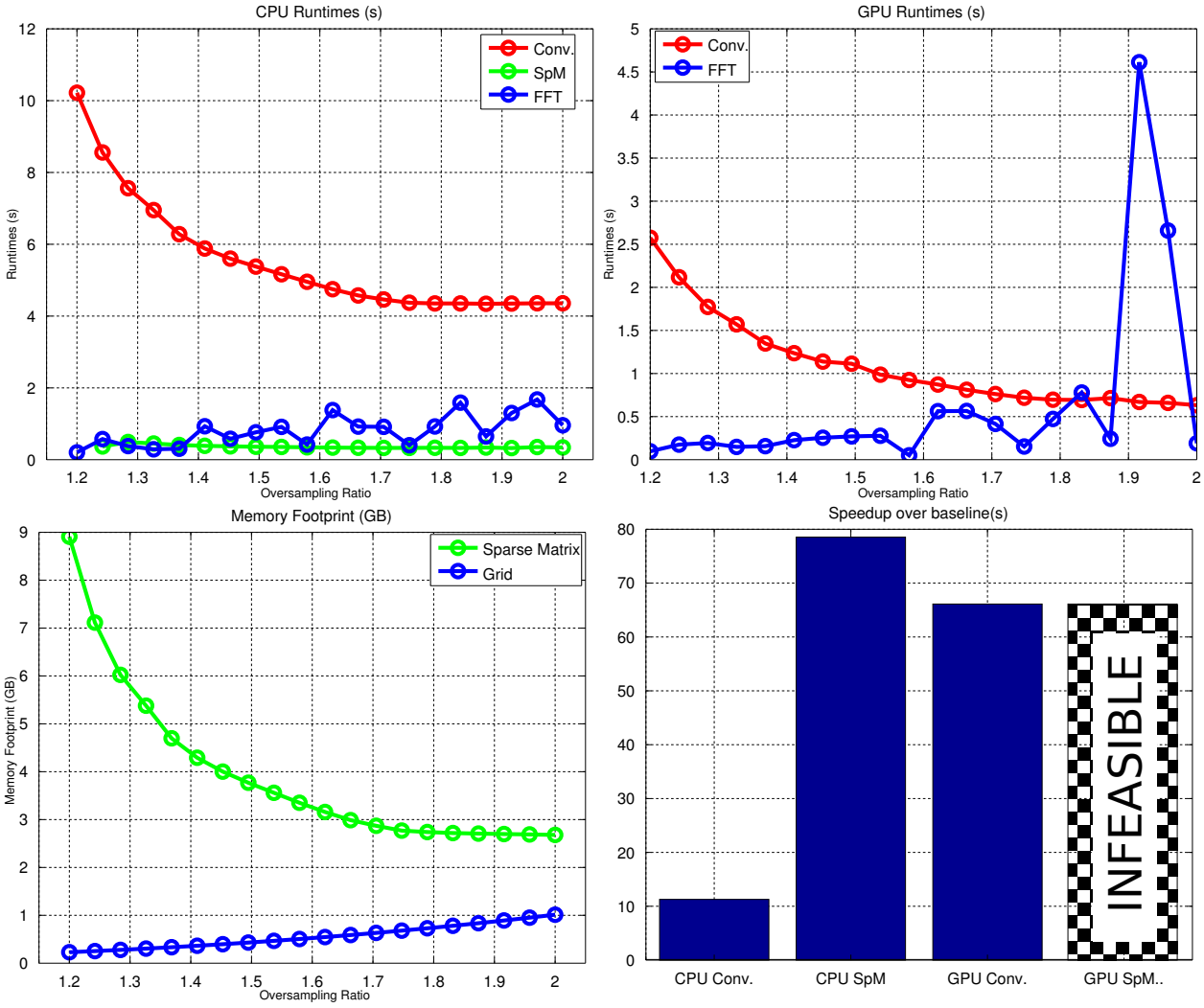

Figure 5.4: nuFFT auto-tuning performance for the 26M-sample 3D Radial trajectory. For description of the plots in the four panels, see Figure 5.3. Note that the sparse matrix implementation is infeasible on the GPU for this large trajectory due to the 3 GB memory capacity of the GPU used in our evaluation. Consequently, the highest-performing implementation uses the CPUs with resampling via matrix-vector multiplication.

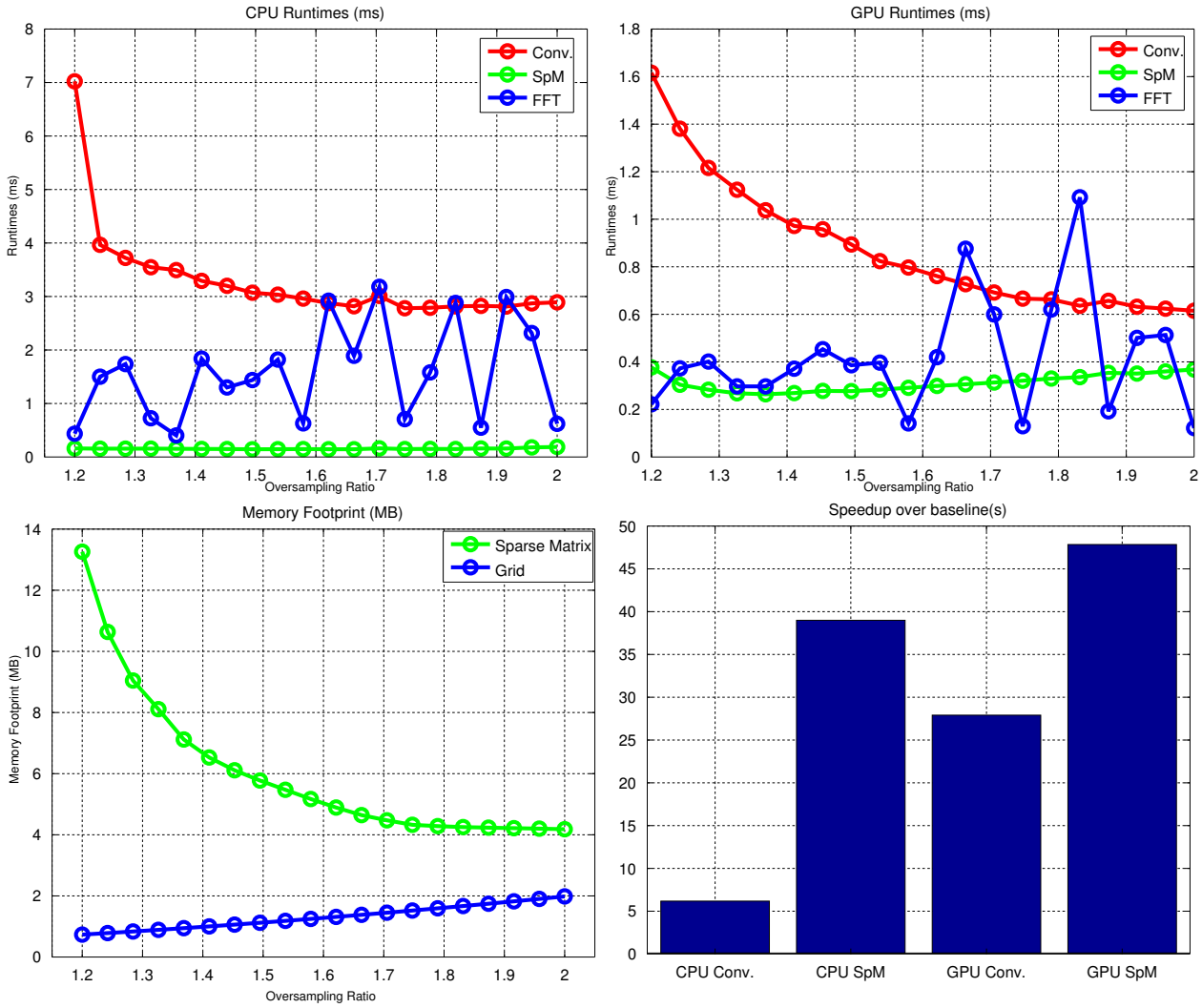

Figure 5.5: nuFFT performance for the 36K-sample 2D Spiral trajectory. For description of the plots in the four panels, see Figure 5.3. Note that runtimes are reported in milliseconds and memory footprints reported in megabytes, whereas in Figures 5.3 and 5.4 they were reported in seconds and Gigabytes.

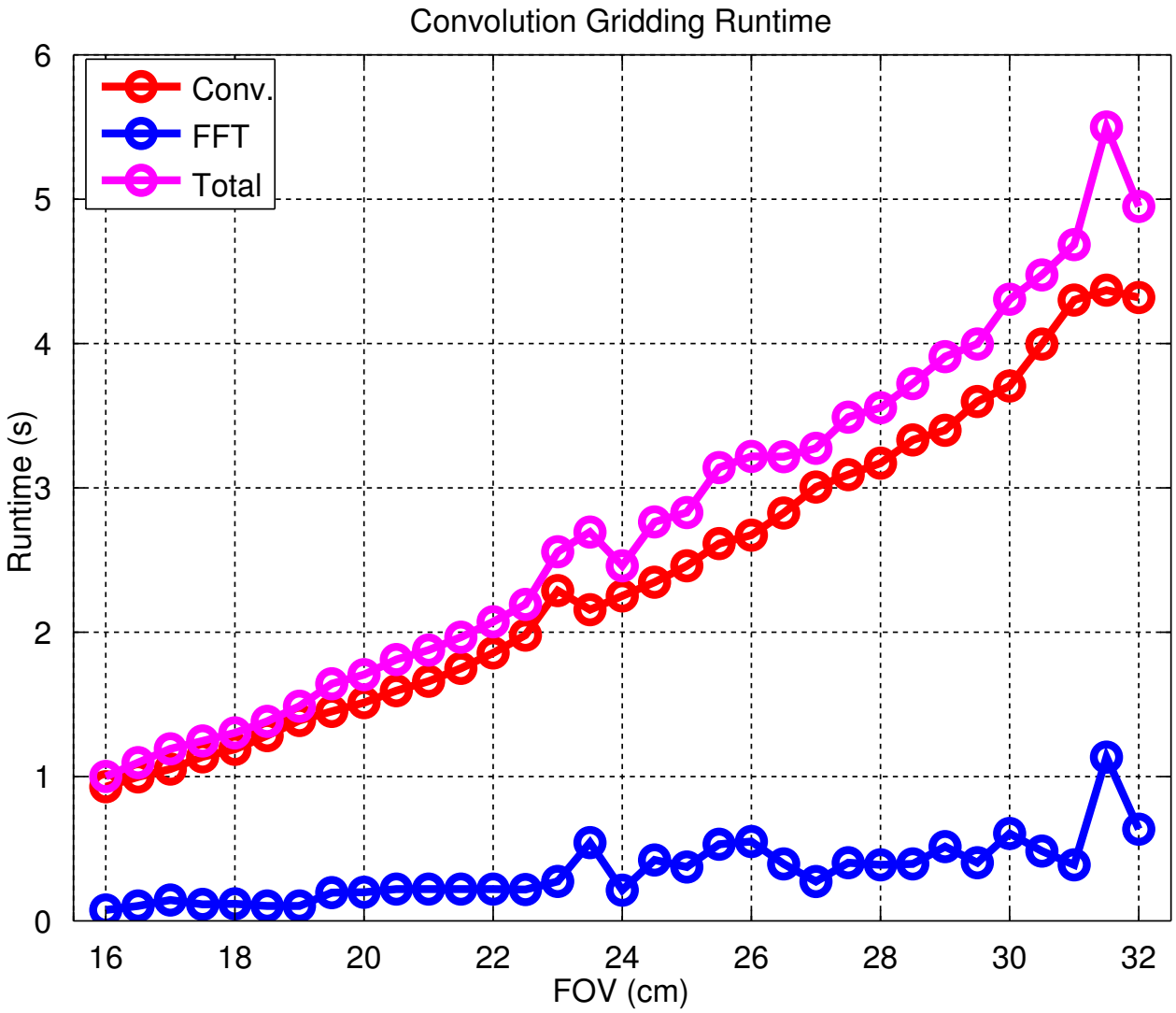

Figure 5.6: Tuned multi-core CPU runtime for Convolution-based nuFFT, for a range of 3D Cones trajectories with isotropic Field-of-View (FOV) varying from 16 cm to 32 cm, all with isotropic 1 mm spatial resolution. In all cases the convolution dominates the overall nuFFT runtime, even though oversampling ratio is typically chosen to be very large, as demonstrated in Figure 5.8.

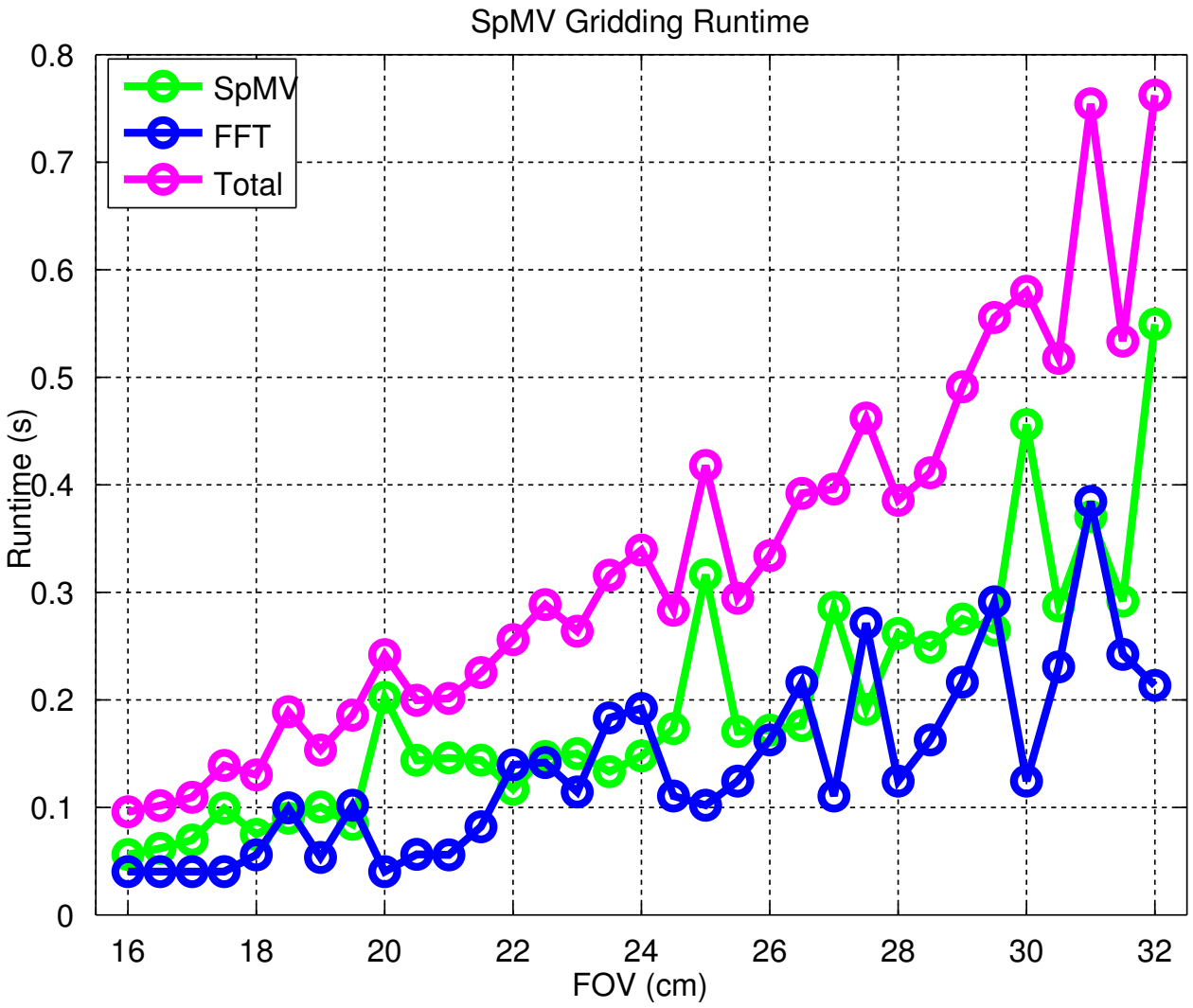

Figure 5.7: Tuned multi-core CPU runtime for Sparse Matrix-based nuFFT, for a range of 3D Cones trajectories with isotropic Field-of-View (FOV) varying from 16 cm to 32 cm, all with isotropic 1 mm spatial resolution. Runtime is well-balanced between the matrix-vector product and the FFT, as relatively low oversampling ratios are chosen by the tuning process (see Figure 5.8). The matrix-vector implementation is consistently much faster than the convolution-based implementation shown in Figure **??**.

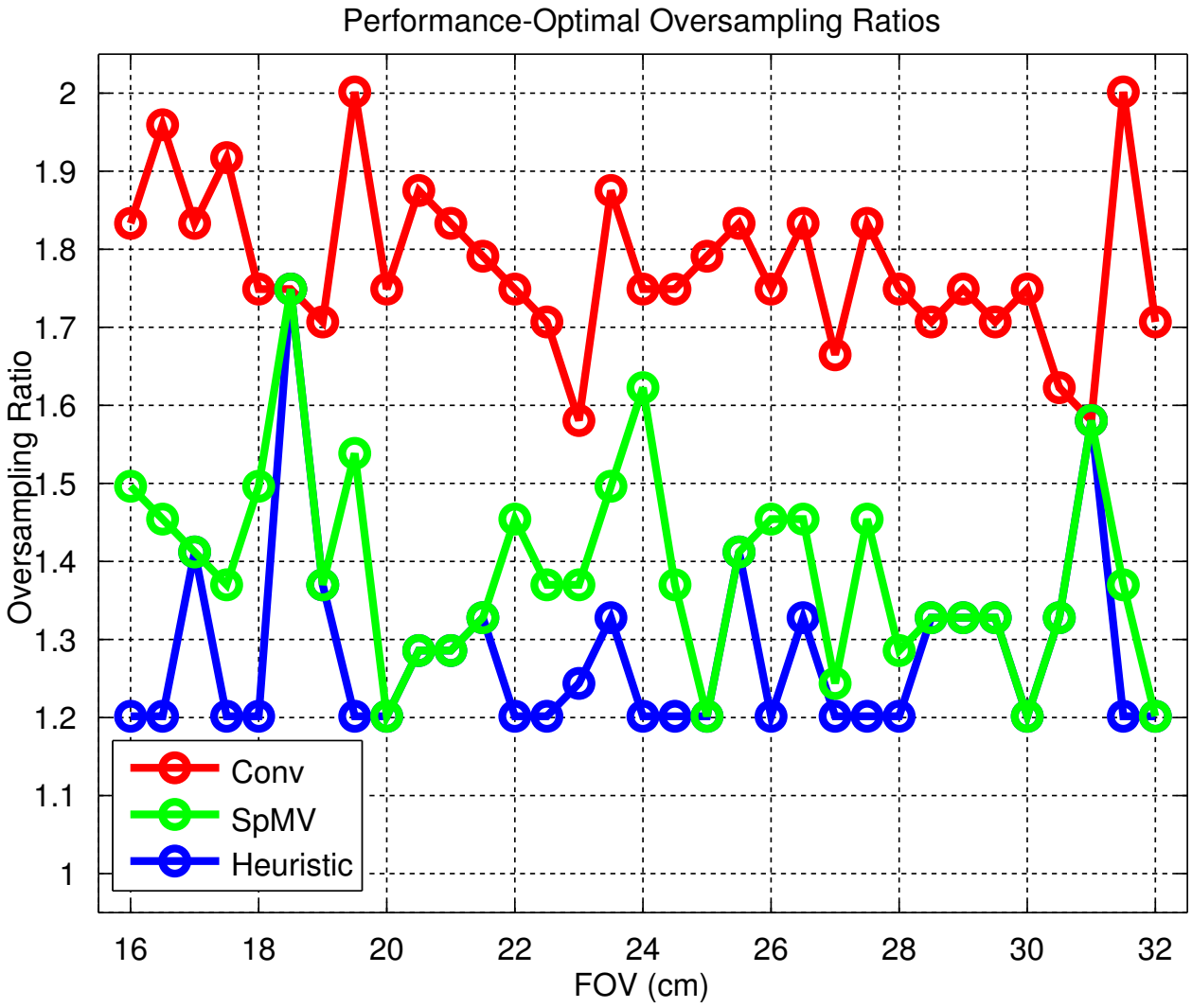

Figure 5.8: Performance-optimal oversampling ratios for the Convolution-based (red points) and Sparse Matrix-based (green points) multi-core CPU implementations of the nuFFTs shown in Figures 5.6 and 5.7, respectively. The blue line shows the alphas selected via a simple heuristic that minimizes FFT runtime, without evaluating sparse matrix-vector runtime. In most cases, the heuristic selects the minimum oversampling ratio considered. However, in some cases FFT performance is pathologically bad for the minimum image matrix size, and FFT performance is improved by choosing a slightly larger grid size.

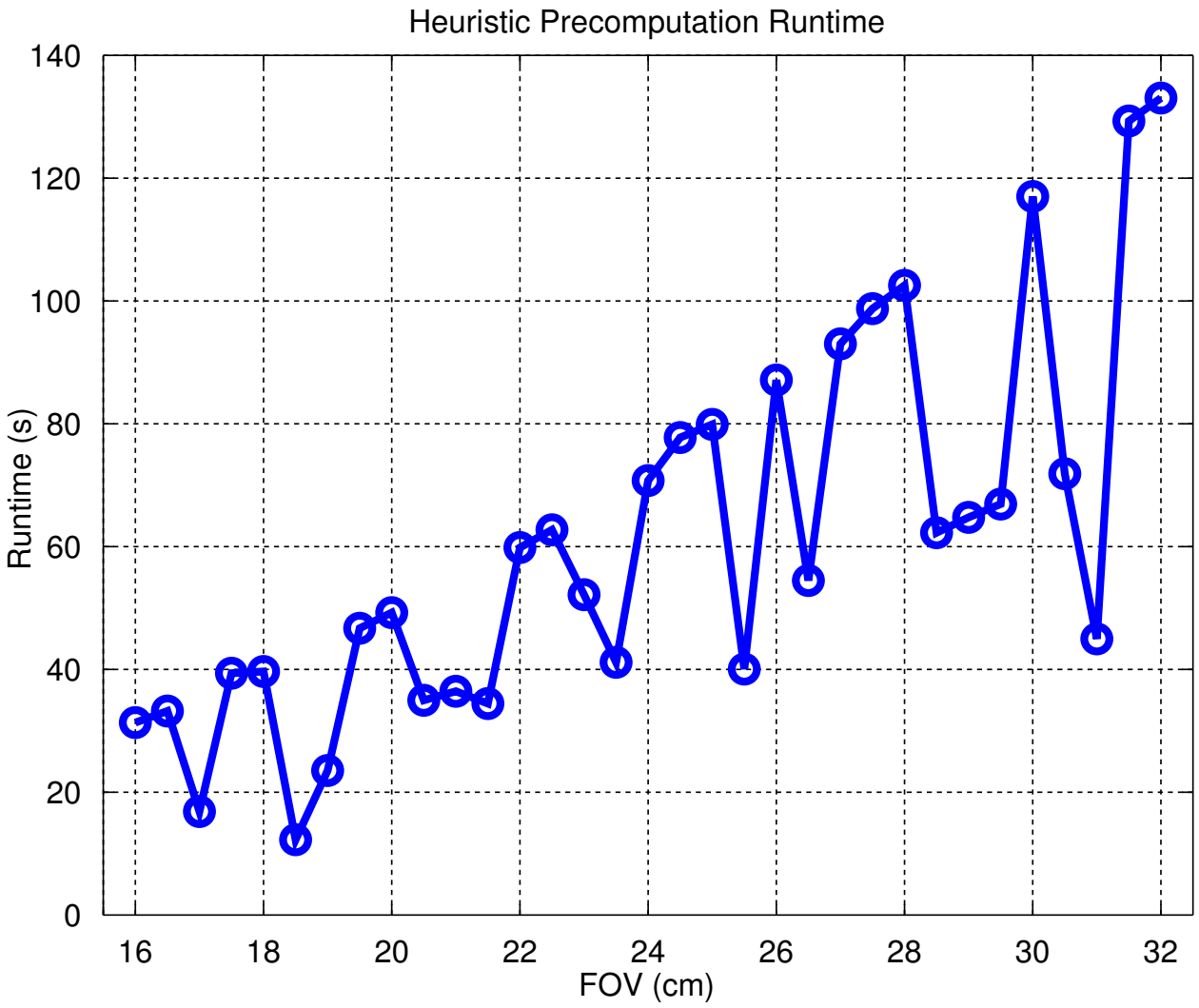

Figure 5.9: Runtime of sparse matrix precomputation for the matrices chosen by the heuristic described in Figure 5.10. Precomputation requires enumerating and sorting all the nonzeros in the sparse matrix. Our current implementation is not optimized, and relies on C++ Standard Template Library (STL) container classes for memory management and sorting routines. Although further performance improvement is desirable for this phase, it can be performed off-line as described in Section 5.6. The 1–2 minute runtimes shown here can easily be overlapped with data acquisition in many MRI contexts.

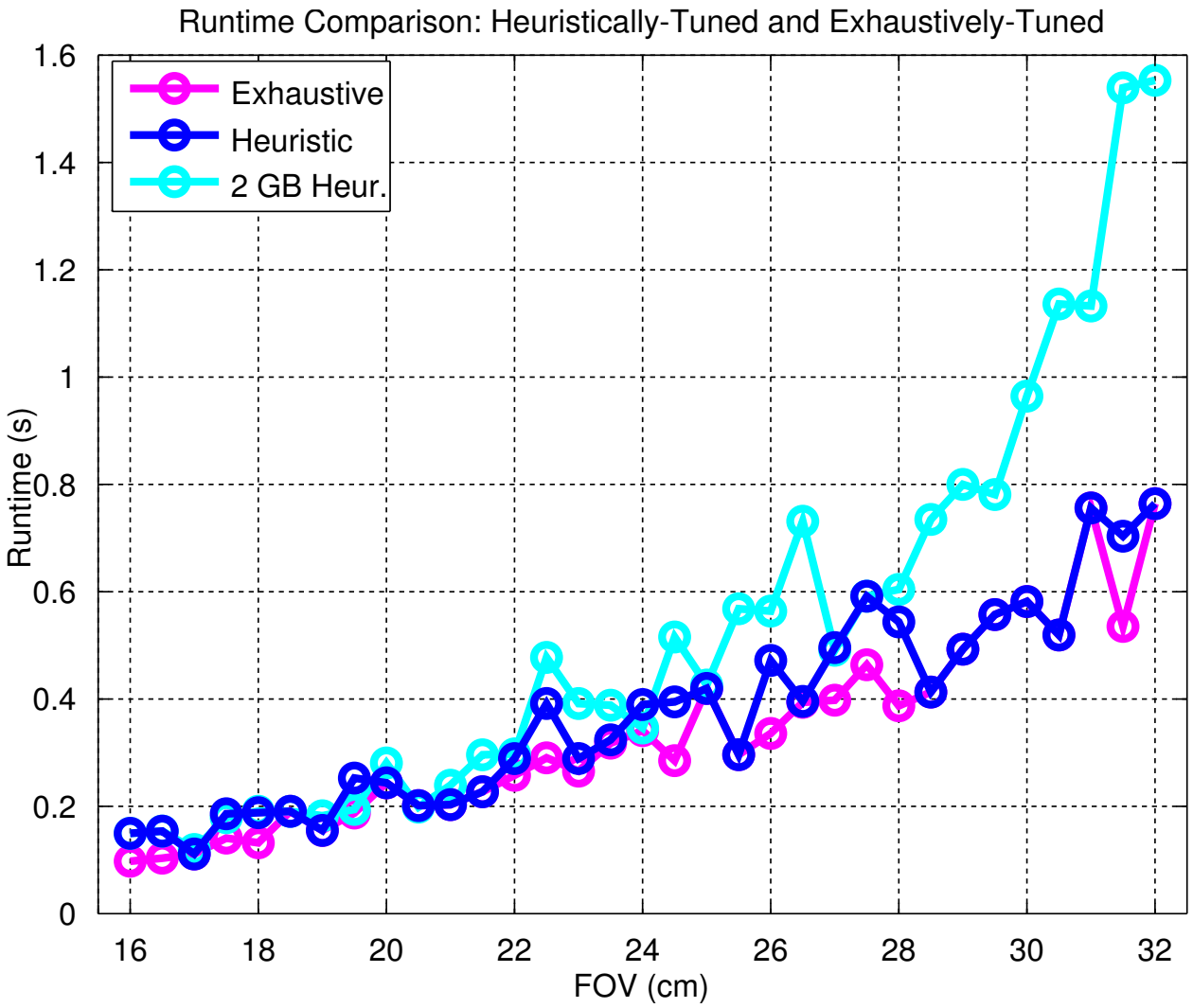

Figure 5.10: Comparison of multi-core CPU nuFFT runtimes resulting from exhaustive tuning (magenta series) and from the inexpensive heuristic (blue series). This heuristic chooses oversampling ratio to minimize FFT runtime and requires computation of only a single sparse matrix during planning. In some applications it may be desirable to limit the size of the sparse matrix, thus restricting the nuFFT to higher oversampling ratios. The heuristic can be modified to choose a ratio only when the resulting sparse matrix would fit within a specified amoutn of memory. The cyan series plots the performance achieved by choosing the fastest FFT runtime subject to a 2 Gigabyte restriction on sparse matrix footprint.

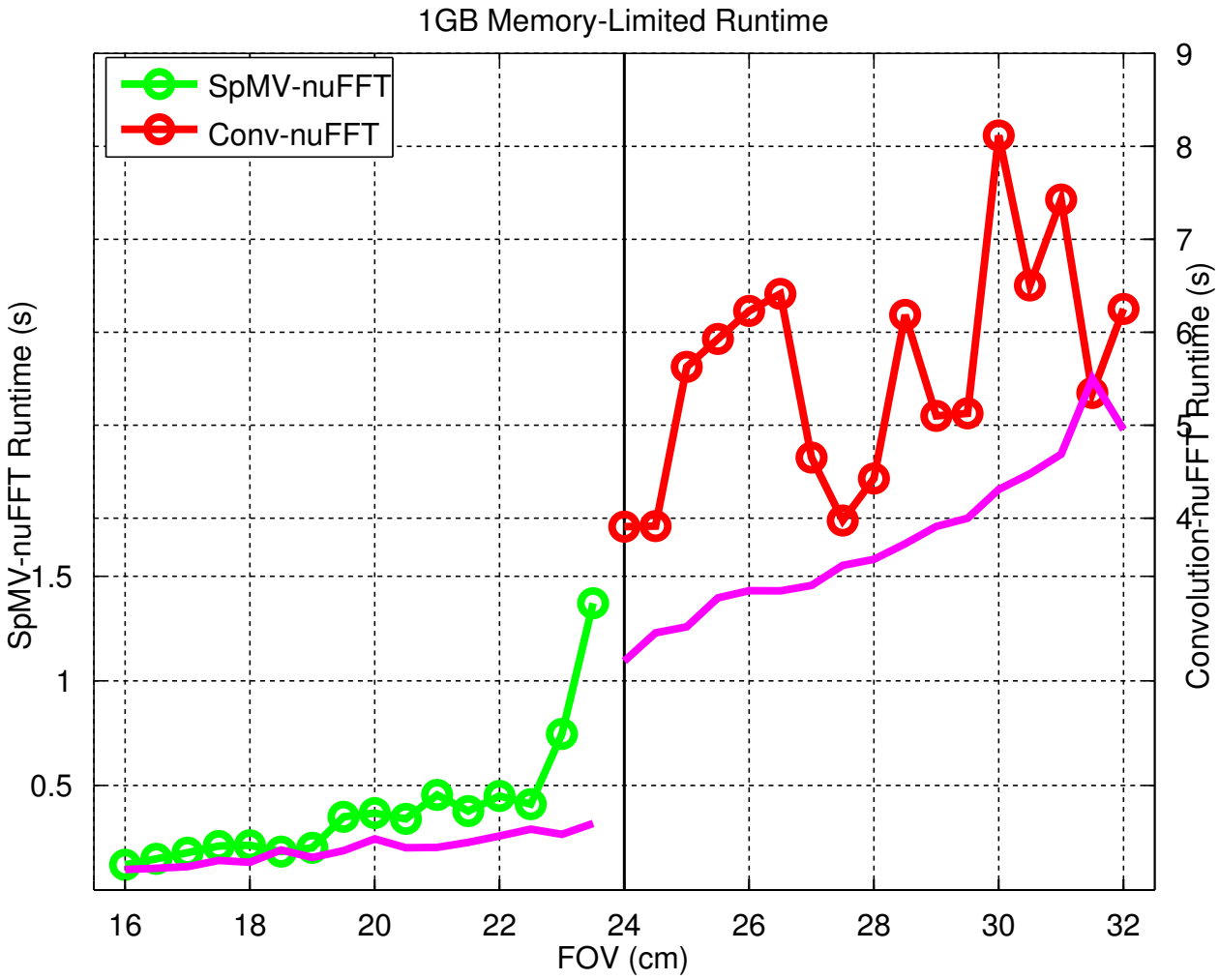

Figure 5.11: Multi-core CPU runtime of heuristically-tuned nuFFT when sparse matrix precomputation is infeasible for some trajectories. In this experiment, memory footprint is limited to 1 gigabyte, which prohibits storing the sparse matrix for all trajectories with FOV larger than 24 cm. When sparse matrix storage is infeasible, grid oversampling ratio is chosen in the range 1.5–2.0 to minimize FFT runtime. The green plot (measured via the left axis) shows nuFFT runtimes for the trajectories whose matrix can be stored in the 1 GB limit, and the red plot (right axis) shows runtimes for the larger trajectories for which no grid oversampling ratio produces a matrix smaller than 1 GB. The magenta line shows the performance achieved by optimal, exhaustive tuning.

Chapter 6

Summary, Conclusion, and Future Work

In this thesis, we have discussed the design and implementation of MRI reconstructions on modern massively parallel processors. Recently proposed iterative algorithms for reconstruction are substantially more computationally intense than their direct predecessors. Although the iterative approaches can enable highly accelerated acquisition and improved image quality, long runtimes resulting from computational intensity are a barrier to their widespread adoption. Modern parallel processing systems, which consist of deeply hierarchical many-core processors and typically combine shared-memory and distributed-memory parallelism, have in recent years become the de-facto standard platform for computationally-intense computing applications. Their widespread adoption has forced a much larger set of application developers to deal with the software development problems of parallelization and performance optimization, problems which had previously only been faced in the domain of scientific computing. In this highly specialized domain, the high cost of the computing platform justifies large software development effort for parallelization and optimization. However, manycore systems have become ubiquitous and inexpensive. It is impractical for all application developers to have expertise simultaneously in their application domain and in computer architecture and performance optimization. This thesis describes a strategy of software development that separates responsibility for low-level performance optimization from higher-level application development.

The Supermarionation architecture, which we described in Chapter 3, composes two design patterns that prescribe implementation strategies for MRI reconstruction algorithms. These algorithms rely on massively parallel implementations of the computationally intense system functions that model the MRI acquisition process. The Geometric Decomposition pattern and the Puppeteer pattern provide solutions to two complementary software development challenges. The Geometric Decomposition pattern describes the several parallelization strategies that can be used to map MRI reconstruction algorithms onto manycore platforms. The Puppeteer pattern prescribes a strategy to decouple the implementation of iterative

reconstruction algorithms from the optimization of the operations that compose the system function. While the optimization of these operations can be left to the implementors of high-performance libraries, the implications of parallelization and data partitioning decisions percolate throughout the software system. The MRI scientist implementing the reconstruction must be aware of these implications, and the software system myst be implemented accordingly.

Our description of the ℓ_1-SPIRiT reconstruction system in Chapter 4 provides a broad-scope view of the application of the Supermarionation architecture. ℓ_1-SPIRiT reconstruction exemplifies the computational difficulties encountered in modern iterative MRI reconstructions. The POCS algorithm estimates the system parameters (i.e. wavelet coefficients of the image) from measurements (samples of k-space) via a system function (composed Wavelet, Fourier, and coil sensitivities). Computation of the operations which compose the system function dominate runtime, and their high-performance implementation is crucial to clinical feasibility of the technique. As a result of our optimization efforts, the ℓ_1-SPIRiT reconstruction has been in deployment for long enough to perform substantial clinical experimentation and evaluation.

The detailed optimization study of the Non-Uniform Fast Fourier Transform (nuFFT) in Chapter 5 provides an in-depth look at one element of the high-performance library for MRI reconstruction. The nuFFT, also called Gridding in MRI applications, is a crucial component of reconstructions for non-Cartesian k-space samplings. The same techniques used to optimize the nuFFT can similarly be used to optimize other computations. For example, the technique of empirical search has successfully been used for the FFT and related signal-processing transforms, sparse matrix-vector multiplication, and dense linear algebra operations.

We have shown that the performance improvements achieved by application of our performance-optimization techniques are sufficient to achieve clinically-feasible runtimes for advanced, iterative MRI reconstructions. Furthermore, our performance-optimized implementation has enabled substantial evaluation of the ℓ_1-SPIRiT reconstruction in a clinical setting [81, 82]. Our reconstruction has been deployed in this setting for over 1.5 years, and used in concert with standard reconstructions in daily clinical diagnostics. Our end-to-end ℓ_1-SPIRiT system requires only a few minutes to reconstruct clinically-sized volumetric datasets, even when using large parallel imaging coil arrays. Our nuFFT implementation vastly improves Gridding runtimes, and by precomputing most information essentially removes the performance problem posed by the resampling convolution. Thus we claim that using the Supermarionation architecture and a well-optimized library of computationally-intense operations, a wide variety of MRI reconstruction approaches can successfully be mapped to manycore reconstruction platforms.

Future work will focus on increasing the generality of the software we have described in this thesis, in order to facilitate further algorithmic innovation in MRI reconstruction. The Supermarionation architecture is broad enough to encompass a wide variety of reconstruction algorithms running on any type of modern high-performance computing system. However,

the implementations we have produced for this thesis are of much narrower scope. For the most part, we have focused on the performance problems inherent in the shared-memory parallel systems that have emerged in the past decade. Our ℓ_1-SPIRiT implementation described in Chapter 4 leverages the distributed memory architecture of multi-GPU systems. However, multi-GPU systems are of much smaller scale than massively parallel distributed memory clusters, the largest of which include tens of thousands of distributed memory nodes. Although very few MRI reconstruction tasks can leverage machines of this scale, in many contexts the limitations of GPU architectures and programming environments prohibit their use. Instead, a distributed memory cluster of moderate scale is more desirable. In this case, the Geometric Decomposition strategy can still guide implementation decisions, although distributed memory data partitioning will require substantial additional programmatic effort. Additionally, our instantiations of the components of Supermarionation software systems are not optimally re-usable. Future work can take advantage of the many opportunities to hide optimization and data-layout decisions via appropriate object oriented language features. In this way, implementations of iterative algorithms and computationally intense operations can be made much more flexible and re-usable. The combination of distributed-memory parallelization and re-usable implementation according to the Supermarionation architecture will greatly ease future research in algorithms for and applications of iterative MRI reconstructions.

Bibliography

[1] Mehmet Akcakaya, Tamer Basha, Warren Manning, Seunghoon Nam, Reza Nezafat, Christian Stehning, and Vahid Tarokh. A GPU implementation of compressed sensing reconstruction of 3D radial (kooshball) acquisition for high-resolution cardiac MRI. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2011.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[5] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, Henk van der Vorst, and Long Restart. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1993.

[6] P.J. Beatty, A Brau, S Chang, S Joshi, Michelich C, Bayram E, T Nelson, R Herfkens, and J Brittain. A method for autocalibrating 2D-accelerated volumetric parallel imaging with clinically practical reconstruction times. In *Proceedincs of the Joint Annual Meeting ISMRM-ESMRMB*, page 1749, 2007.

[7] P.J. Beatty, D.G. Nishimura, and J.M. Pauly. Rapid gridding reconstruction with a minimal oversampling ratio. *IEEE Trans Med Imaging*, pages 799–808, Jun.

[8] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183, 2009.

[9] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[10] M.A. Bernstein, K.F. King, and X.J. Zhou. *Handbook of MRI pulse sequences*. Elsevier Academic Press, 2004.

[11] Berkin Bilgic, Vivek K Goyal, and Elfar Adalsteinsson. Multi-contrast reconstruction with Bayesian compressed sensing. *Magnetic Resonance in Medicine*, 000:1–15, 2011.

[12] Kai Tobias Block, Martin Uecker, and Jens Frahm. Undersampled radial MRI with multiple coils. Iterative image reconstruction using a total variation constraint. *Magn Reson Med*, 57(6):1086–98, Jun 2007.

[13] RW Buccigrossi and EP Simoncelli. Image compression via joint statistical characterization in the wavelet domain. *IEEE Trans. Image Processing*, 8:1688–1701, 1999.

[14] Martin Buehrer, Klaas P Pruessmann, Peter Boesiger, and Sebastian Kozerke. Array compression for MRI with large coil arrays. *Magnetic Resonance in Medicine*, 57(6):1131–1139, 2007.

[15] Martin Buehrer, Klaas P Pruessmann, Peter Boesiger, and Sebastian Kozerke. Array compression for MRI with large coil arrays. *Magn Reson Med*, 57(6):1131–9, 2007.

[16] E.J. Candès, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Transactions on Information Theory*, 52:489–509, 2006.

[17] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[18] Ching-Hua Chang and Jim Ji. Compressed sensing MRI with multichannel data using multicore processors. *Magn Reson Med*, 64(4):1135–9, Oct 2010.

[19] Scott Shaobing Chen, David L Donoho, and Michael A Saunders. Atomic decomposition by basis pursuit. *SIAM Review*, 43(1):129, 2001.

[20] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *Siam Review*, December 2008.

[21] Kauskik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Supercomputing*, November 2008.

[22] I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Comm. Pure Applied Mathematics*, 57:1413 – 1457, 2004.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[24] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[25] David L. Donoho, Yaakov Tsaig, Iddo Drori, and Jean luc Starck. Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit. Technical report, Department of Statistics, Stanford University, 2006.

[26] D.L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, 52:1289–1306, 2006.

[27] D.L. Donoho and I.M. Johnstone. Ideal spatial adaptation via wavelet shrinkage. *Biometrika*, 81:425–455, 1994.

[28] H. Eggers, T. Knopp, and D. Potts. Field inhomogeneity correction based on gridding reconstruction for magnetic resonance imaging. *Medical Imaging, IEEE Transactions on*, 26(3):374 –384, march 2007.

[29] M Elad, B Matalon, and M Zibulevsky. Coordinate and subspace optimization methods for linear least squares with non-quadratic regularization. *Applied and Computational Harmonic Analysis*, 23(3):346–367, 2007.

[30] Asanovic et al. The landscape of parallel computing research: a view from berkeley. Technical report, EECS Department, University o California, berkeley, Dec 2006.

[31] J.A. Fessler and B.P. Sutton. A min-max approach to the multidimensional nonuniform FFT: application to tomographic image reconstruction. In *Image Processing, 2001. Proceedings. 2001 International Conference on*, volume 1, pages 706 –709 vol.1, 2001.

[32] M.A.T. Figueiredo and R.D. Nowak. An EM algorithm for wavelet-based image restoration. *Image Processing, IEEE Transactions on*, 12(8):906 – 916, August 2003.

[33] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, feb. 2005.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[35] Gary H. Glover and Norbert J. Pelc. Method for correcting image distortion due to gradient nonuniformity, 05 1986.

[36] Anthony Gregerson. Implementing fast MRI Gridding on GPUs via Cuda. Technical report, University of Wisconsin, Madison, 2008.

[37] Mark A Griswold, Peter M Jakob, Robin M Heidemann, Mathias Nittka, Vladimir Jellus, Jianmin Wang, Berthold Kiefer, and Axel Haase. Generalized autocalibrating partially parallel acquisitions (GRAPPA). *Magn Reson Med*, 47(6):1202–10, 2002.

[38] Paul T Gurney. *Magnetic resonance imaging using a 3D cones k-space trajectory*. PhD thesis, Stanford University, 2007.

[39] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[40] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[41] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[42] Michael D Hope, Thomas A Hope, Stephen E S Crook, Karen G Ordovas, Thomas H Urbania, Marc T Alley, and Charles B Higgins. 4D flow CMR in assessment of valve-related ascending aortic disease. *JACC Cardiovascular imaging*, 4(7):781–787, 2011.

[43] Feng Huang, Yunmei Chen, Wotao Yin, Wei Lin, Xiaojing Ye, Weihong Guo, and Arne Reykowski. A rapid and robust numerical algorithm for sensitivity encoding with sparsity constraints: self-feeding sparse SENSE. *Magn Reson Med*, 64(4):1078–88, Oct 2010.

[44] J.I. Jackson, C.H. Meyer, D.G. Nishimura, and A. Macovski. Selection of a convolution function for Fourier inversion using gridding [computerised tomography application]. *Medical Imaging, IEEE Transactions on*, 10(3):473 –478, sep 1991.

[45] IBM journal of Research and Development staff. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52:199–220, January 2008.

[46] Jens Keiner, Stefan Kunis, and Daniel Potts. Using NFFT 3—a software library for various nonequispaced fast fourier transforms. *ACM Transactions on Mathematical Software*, 36(4):Article 19, 1 — 30, 2009.

[47] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[48] Daehyun Kim, Joshua D Trzasko, Mikhail Smelyanskiy, Clifton R Haider, Armando Manduca, and Pradeep Dubey. High-performance 3D compressive sensing MRI reconstruction. *Conf Proc IEEE Eng Med Biol Soc*, 2010:3321–4, 2010.

[49] Florian Knoll, Christian Clason, Kristian Bredies, Martin Uecker, and Rudolf Stollberger. Parallel imaging with nonlinear reconstruction using variational penalties. *Magn Reson Med*, Jun 2011.

[50] Tobias Knopp, Stefan Kunis, and Daniel Potts. A note on the iterative MRI reconstruction from nonuniform k-space data. *International Journal of Biomedical Imaging*, 2007:24727.

[51] Peng Lai, Michael Lustig, Anja CS. Brau, Shreyas Vasanawa la, Philip J. Beatty, and Marcus Alley. Efficient ℓ_1-SPIRiT reconstruction (ESPIRiT) for highly accelerated 3D volumetric MRI with parallel imaging and compressed sensing. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, page 345, 2010.

[52] Dong Liang, Bo Liu, Jiunjie Wang, and Leslie Ying. Accelerating SENSE using compressed sensing. *Magn Reson Med*, 62(6):1574–84, Dec 2009.

[53] Bo Liu, Kevin King, Michael Steckner, Jun Xie, Jinhua Sheng, and Leslie Ying. Regularized sensitivity encoding (SENSE) reconstruction using Bregman iterations. *Magnetic Resonance in Medicine*, 61(1):145–152, 2009.

[54] M. Lustig, M. Alley, S. Vasanawala, D.L. Donoho, and J.M. Pauly. ℓ_1-SPIRiT: Autocalibrating parallel imaging compressed sensing. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, page 379, 2009.

[55] Michael Lustig, David L. Donoho, and John Mark Pauly. Sparse MRI: The application of compressed sensing for rapid MR imaging. *Magn Reson Med*, 58(6):1182–1195, 2007.

[56] Michael Lustig, Seung-Jean Kim, and John M Pauly. A fast method for designing time-optimal gradient waveforms for arbitrary k-space trajectories. *IEEE Transactions on Medical Imaging*, 27(6):866–873, 2008.

[57] Michael Lustig and John M. Pauly. SPIRiT: Iterative self-consistent parallel imaging reconstruction from arbitrary k-space. *Magnetic Resonance in Medicine*, 64(2):457–471, 2010.

[58] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[59] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[60] M. Murphy, K. Keutzer, S. Vasanawala, and M. Lustig. Clinically feasible reconstruction time for ℓ_1-SPIRiT parallel imaging and compressed sensing MRI. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, page 4854, 2010.

[61] D. C. Noll, D. G. Nishimura, and A. Macovski. Homodyne detection in magnetic resonance imaging. *Medical Imaging, IEEE Transactions on*, 10(2):154–163, 1991.

[62] Nvidia. Compute Unified Device Architecture (Cuda). `http://www.nvidia.com/object/cuda_get.html`. [Online; accessed 25 July, 2011].

[63] Whitepaper Nvidia, Next Generation, and Cuda Compute. NVIDIAs next generation CUDA compute architecture. *ReVision*, pages 1–22, 2009.

[64] Nady Obeid, Ian Atkinson, Keith Thulborn, and Wen-Mei Hwu. GPU-accelerated gridding for rapid reconstruction of non-Cartesian MRI. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2011.

[65] Ricardo Otazo, Daniel Kim, Leon Axel, and Daniel K Sodickson. Combination of compressed sensing and parallel imaging for highly accelerated first-pass cardiac perfusion MRI. *Magn Reson Med*, 64(3):767–76, Sep 2010.

[66] K P Pruessmann, M Weiger, P Börnert, and P Boesiger. Advances in sensitivity encoding with arbitrary k-space trajectories. *Magn Reson Med*, 46(4):638–51, Oct 2001.

[67] K P Pruessmann, M Weiger, M B Scheidegger, and P Boesiger. SENSE: sensitivity encoding for fast MRI. *Magn Reson Med*, 42(5):952–62, Nov 1999.

[68] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[69] S. Roujol, B. D. de Senneville, E. Vahala, T. S. Sorensen, C. Moonen, and M Ries. Online real-time reconstruction of adaptive TSENSE with commodity CPU/GPU hardware. *Magnetic Resonance in Medicine, 62:1658-1664*, 2009.

[70] Damian Rouson, Jim Xia, and Helgi Adalsteinsson. Design patterns for multiphysics modeling in Fortran 2003 and C++. *ACM Transactions on Mathematical Software*, 37(1):3:1–3:30, January 2010.

[71] D K Sodickson and W J Manning. Simultaneous acquisition of spatial harmonics (SMASH): fast imaging with radiofrequency coil arrays. *Magn Reson Med*, 38(4):591–603, Oct 1997.

[72] Thomas Sangild Sorensen, Claudia Prieto, David Atkinson, Michael Schacht Hansen, and Tobias Schaeffter. GPU accelerated iterative SENSE reconstruction of radial phase encoded whole-hyeart MRI. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2010.

[73] T.S. Sorensen, T. Schaeffter, K.O. Noe, and M.S. Hansen. Accelerating the nonequispaced fast fourier transform on commodity graphics hardware. *Medical Imaging, IEEE Transactions on*, 27(4):538 –547, april 2008.

[74] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Accelerating advanced MRI reconstructions on GPUs. In *Proceedings of the 5th conference on Computing Frontiers*, 2008.

[75] David S. Taubman and Michael W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice.* Kluwer International Series in Engineering and Computer Science., 2002.

[76] K. R. Thulborn, J. C. Waterton, P. M. Matthews, and G. K. Radda. Oxygenation dependence of the transverse relaxation time of water protons in whole blood at high field. *Biochem. Biophys. Acta*, 714:265–270, 1982.

[77] Joel A. Tropp, Anna, and C. Gilbert. Signal recovery from random measurements via Orthogonal Matching Pursuit. *IEEE Trans. Inform. Theory*, 53:4655–4666, 2007.

[78] Joshua D Trzasko, Clifton R Haider, Eric A Borisch, Norbert G Campeau, James F Glockner, Stephen J Riederer, and Armando Manduca. Sparse-CAPR: Highly accelerated 4D CE-MRA with parallel imaging and nonconvex compressive sensing. *Magn Reson Med*, 66(4):1019–32, Oct 2011.

[79] Martin Uecker, Thorsten Hohage, Kai Tobias Block, and Jens Frahm. Image reconstruction by regularized nonlinear inversion–joint estimation of coil sensitivities and image content. *Magn Reson Med*, 60(3):674–82, Sep 2008.

[80] Martin Uecker, Shuo Zhang, and Jens Frahm. Nonlinear inverse reconstruction for real-time MRI of the human heart using undersampled radial FLASH. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2010.

[81] Shreyas S Vasanawala, Marcus T Alley, Brian A Hargreaves, Richard A Barth, John M Pauly, and Michael Lustig. Improved pediatric MR imaging with compressed sensing. *Radiology*, 256(2):607–16, Aug 2010.

[82] S.S. Vasanawala, M.J. Murphy, M.T. Alley, P. Lai, K. Keutzer, J.M. Pauly, and M. Lustig. Practical parallel imaging compressed sensing MRI: Summary of two years of experience in accelerating body MRI of pediatric patients. In *Proceedings of IEEE International Symposium on Biomedical Imaging*, pages 1039 –1043, Chicago, 2011.

[83] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[84] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[85] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Supercomputing*, November 2007.

[86] Stephen J. Wright, Robert D. Nowak, and Mário A. T. Figueiredo. Sparse reconstruction by separable approximation. *Trans. Sig. Proc.*, 57:2479–2493, July 2009.

[87] Bing Wu, Rick P Millane, Richard Watts, and Philip J Bones. Prior estimate-based compressed sensing in parallel mri. *Magn Reson Med*, 65(1):83–95, Jan 2011.

[88] Xiao-Long Wu, Jiading Gai, Fan Lam, Maojing Fu, Justin Haldar, Yue Zhuo, Zhi-Pei Liang, Wen mei Hwu, and Bradley Sutton. IMPATIENT MRI: Illinois massively parallel acceleration toolkit for image reconstruction with enhanced throughput in MRI. In *Proceedings of the IEEE International Symposium on Biomedical Imaging (ISBI)*, 2011.

[89] Allen Y. Yang, Arvind Ganesh, Zihan Zhou, Shankar Sastry, and Yi Ma. A review of fast l1-minimization algorithms for robust face recognition. *CoRR*, abs/1007.3753, 2010.

[90] Leslie Ying and Jinhua Sheng. Joint image reconstruction and sensitivity estimation in SENSE (JSENSE). *Magn Reson Med*, 57(6):1196–202, Jun 2007.

[91] Tao Zhang, Michael Lustig, Shreyas Vasanawala, and John Pauly. Array compression for 3D Cartesian sampling. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, page 2857, 2011.

[92] Tiejun Zhao and Xiaoping Hu. Iterative GRAPPA (iGRAPPA) for improved parallel imaging reconstruction. *Magnetic Resonance in Medicine*, 59(4):903–907, 2008.

[93] Yue Zhuo, Xiao-Long Wu, Justin P. Haldar, Wen-Mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton. Multi-GPU implementation for iterative MR image reconstruction with field correction. In *Proceedings of the International Society for Magnetic Resonance in Medicine*, 2010.