Domain-Specific Techniques for High-Performance Computational Image Reconstruction

Michael Driscoll

Acknowledgement

This research was supported in part by Department of Energy, Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory contract DEAC02-05CH11231 and UC Berkeley contract DE-SC0008700. This research used resources of the National Energy Research Scientific Computing Center, and the Oak Ridge Leadership Computing Facility, all supported by the Office of Science of the U.S. Department of Energy under contracts No. DE-AC02-05CH11231 and DE-AC05-00OR22725, respectively.

Domain-Specific Techniques for High-Performance Computational Image Reconstruction

by

Michael Driscoll

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Katherine Yelick, Co-chair
Professor Armando Fox, Co-chair
Professor Michael Lustig
Professor Steven Conolly

Fall 2018

Domain-Specific Techniques for High-Performance Computational Image Reconstruction

Abstract

Domain-Specific Techniques for High-Performance Computational Image Reconstruction

by

Michael Driscoll

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Katherine Yelick, Co-chair

Professor Armando Fox, Co-chair

The widespread emergence of parallel computers in the last decade has created a substantial programming challenge for application developers who wish to attain peak performance for their applications. Parallel programming requires significant expertise, and programming tools—general-purpose languages, compilers, libraries, etc.—have had limited success in hiding the complexity of parallel architectures. Furthermore, the parallel programming burden is likely to increase as processor core counts grow and memory hierarchies become deeper and more complex.

The challenge of delivering productive high-performance computing is especially relevant to computational imaging. One technique in particular, iterative image reconstruction, has emerged as a prominent technique in medical and scientific imaging because it offers enticing application benefits. However, it often demands high-performance implementations that can meet tight application deadlines, and the ongoing development of the iterative reconstruction techniques discourages ad-hoc performance optimization efforts.

This work explores productive techniques for implementing fast image reconstruction codes. We present a domain-specific programming language that is expressive enough to represent a variety of important reconstruction problems, but restrictive enough that its programs can be analyzed and transformed to attain good performance on modern multi-core, many-core and GPU platforms. We present case studies from magnetic resonance imaging (MRI), ptychography, magnetic particle imaging, and microscopy that achieve up to 90% of peak performance. We extend our work to the distributed-memory setting for an MRI reconstruction task. There, our approach gets perfect strong scaling for reasonable machine sizes, and sets the best-known reconstruction time for our particular reconstruction task. The results indicate that a domain-specific language can be successful in hiding much of the complexity of implementing fast reconstruction codes.

For Samanta.

Contents

List of Figures

List of Tables

List of Symbols

ADMM	Alternating Direction Method of Multipliers.
AXBY	Vector-vector multiplication ($\alpha X + \beta Y$).
BART	Berkeley Advanced Reconstruction Toolbox.
BLAS	Basic Linear Algebra Subroutines.
CPU	central processing unit.
DDR4	double data-rate fourth generation.
DRAM	dynamic random access memory.
DSL	domain-specific language.
FFT	Fast Fourier Transform.
FISTA	Fast Iterative Shrinkage-Thresholding Algorithm.
FLOP	Floating-Point Operation.
GB	Gigabytes.
GEMM	general matrix-matrix multiplication.
HPC	High-Performance Computing.
ISTA	Iterative Shrinkage-Thresholding Algorithm.
KB	Kilobytes.
KNL	Knights Landing.
LLR	Locally Low-Rank.
MB	Megabytes.
MCDRAM	multi-channel dynamic random access memory.
MIRT	Michigan Image Reconstruction Toolbox.
MPI	Message Passing Interface.
MRI	Magnetic Resonance Imaging.
NERSC	the National Energy Research Scientific Computing Center.
NUFFT	non-uniform Fast Fourier Transform.
NUMA	non-uniform memory access.
OneMM	Multiplicate by a matrix of ones.
PICS	Parallel-Imaging Compressed-Sensing Reconstruction.
PICS-DM	Distributed-Memory PICS.

SENSE	SENSitivity Encoding.
SpMM	Sparse Matrix times Dense Matrix Multiplication.
SpMV	Sparse Matrix-Vector Multiplication.
TB	Terabytes.
TCP	Transfer Control Protocol.
UWUTE	University of Wisconsin Ultra-short Time Echo.

Acknowledgments

This dissertation would not have been possible without the guidance and support of my advisors, Kathy Yelick and Armando Fox. The breadth of academic freedom that they provided was both exciting and challenging. I am ever grateful for their insightful feedback and the way it has shaped me as a researcher. I would also like to thank Miki Lustig and Steve Conolly for serving on my qualifying exam committee, and for their assistance in guiding my research and preparing this dissertation.

I had the privilege of collaborating with many people over my years at Berkeley. Evangelos Georganas and Penporn Koanantakool, my academic siblings, provided years of fruitful discussions about our work. Jim Demmel has been a gracious advisor-in-law and is always eager to consult on research and beyond. I am grateful to the members of the Bebop research group, the ParLab/Aspire Labs, the UPC group at LBNL, the Berkeley Center for Computational Imaging, the research groups at Disney Animation and Pixar Animation Studios, and the Rose Compiler Group at LLNL. I am honored to have been included in so many efforts. I would also like to extend my thanks to the always-helpful support staff at these institutions. Their efforts truly allowed us to focus on the research agenda at hand.

Finally, I would like to thank my friends and family for supporting me through the program. My time in Berkeley has been the best of my life, and I look forward to many years of friendship, love, and community.

This research was supported in part by Department of Energy, Office of Science, Advanced Scientific Computing Research X-Stack Program under Lawrence Berkeley National Laboratory contract DEAC02-05CH11231 and UC Berkeley contract DE-SC0008700. This research used resources of the National Energy Research Scientific Computing Center, and the Oak Ridge Leadership Computing Facility, all supported by the Office of Science of the U.S. Department of Energy under contracts No. DE-AC02-05CH11231 and DE-AC05-00OR22725, respectively. The views and conclusions in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Chapter 1

Introduction

A foremost goal of high-performance computing research is performance portability—the capability of an application to attain good performance across a variety of computational platforms. This challenge manifests in a variety of situations that span the spectrum of processing power. Some programmers wish to develop codes on their laptops and later scale them to supercomputers. Others wish to migrate codes from one generation of a processor to the next. Still others wish to run their codes on a variety of processors from the same performance class, often because they support a broad group of users with various hardware platforms. Porting applications between platforms can require substantial effort, especially when performance is only attainable by adapting the application to leverage the idiosyncrasies of a particular platform.

Yet, performance is often at odds with portability due to economic constraints on software development. A programmer with a fixed time budget can either spend his or her time optimizing the application for a particular platform, or porting it to multiple platforms and sacrificing some performance. Such tasks require expertise both in the application and performance engineering domains, hence there are few programmers qualified to develop such codes and their time is a scarce resource. This economic view suggests an additional metric of *productivity* to characterize a given software development strategy in terms of the performance attained per unit of development effort. A highly productive strategy makes it easy to write portable and performant software with minimal effort. Given the scarcity of performance engineering expertise, it is critical to consider productive designs when developing fast, portable software.

Historically, programming systems—languages, compilers, runtimes, etc.—have been reasonably successful in providing both portability and performance. In the last decade, the end of the microprocessor frequency scaling regime has resulted in an explosion of parallel computer architectures with increasing complex memory hierarchies. General-purpose programming systems have largely failed to continue delivering performance, instead shifting the burden of discovering parallelism and managing the memory hierarchy to the programmer. To hide much of this complexity from application developers, some tool developers have adopted a *domain-specific* approach that yields portable performance for a particular problem domain. By restricting what notions can be expressed to those in a particular domain, libraries and tools can once again deliver portable performance. This often takes the form of software libraries, like the Basic Linear Algebra Subroutines (BLAS), or domain-specific programming languages.

Software libraries typically hide performance complexity, but they have their shortcomings. While libraries such as those for dense linear algebra excel at large, regular operations that saturate the underlying arithmetic resources, they can fail to achieve good performance in the presence of 1) variation in the structure of programs, or 2) variation in the structure of data. Some program structures—the way library components are assembled together—admit programs transformations that reduce memory traffic or enable parallelism, but these transformations are rarely discovered and applied by general purpose tools. Similarly, data passed to a program may vary in its structure, suggesting different evaluation schemes, but libraries typically implement only one scheme. Performing the necessary transformations to leverage variations in program structure and data structure is necessary to achieve high efficiency, but it's laborious and hence we seek an automated solution.

This thesis explores a domain-specific software architecture that enables productive development of performance portable codes. We make our study concrete by choosing a particular domain—computational image reconstruction—and assessing our approach on applications in that domain.

Application Motivation

Modern approaches to computational image reconstruction demand fast, portable, easy-to-write reconstruction codes. Furthermore, the demand is increasing as data rates grow and new imaging technologies are developed. In a research setting, a productive programming environment allows scientists to focus on new imaging algorithms, rather than their implementations, and accelerates the algorithmic design cycle in the short term, and the rate of scientific discovery in the long term. Performance portability further eases the porting of satisfactory codes from research to production settings since the codes needn't be scaled to new hardware.

Independent of portability concerns, fast reconstruction is highly desirable in scientific and medical settings. Fast medical image reconstruction can dramatically change clinical practices by making certain imaging modalities more applicable to large portions of the patient population. For scientific imaging, fast reconstruction means quicker iterations and early detection of problems in the experiment setup. This is especially important at large, shared science facilities like the Advanced Light Source, where time on the beamline is limited, must be booked well in advance, often involves travel to the facility, and can generate a substantial amount of image data to be reconstructed.

Performance Motivation

This work builds on high performance computing research. An persistent question for high-performance computing research is: how do we architect software to minimize time-to-solution across the development/deployment lifecycle? Any solution necessarily makes some trade-offs between productivity and performance, hence we seek a balance between them. Assuming a code worthy of optimization—a reasonably productive software architecture and certifiable demand for better performance—we are left with two avenues for improving time-to-solution:

- Algorithmic improvements to reduce the volume of work required to solve a problem, thus minimizing "work units."
- Performance improvements to increase the efficiency of the computational platform, thus maximizing "work units per second". This can be accomplished by re-ordering computation and/or allowing redundant computation to minimize data communication costs.

Algorithmic improvements are almost universally more profitable to pursue. This concept even appears in Berkeley's introductory computer science class when students explore how to compute the n-th Fibonacci number. A naive recursive algorithm requires an exponential stack space and operations ("time"); a memoized version linear space and time; and a two-by-two matrix formulation constant space and linear time. In simulation of human hair for animated feature films [30], it's sufficient and cheaper to apply approximation algorithms that model the dynamics of hair with an invisible mass-spring system than to accurately model its structure and molecular dynamics.

At some point, however, opportunities cease to exist for low-hanging algorithmic improvements. In physical simulation, there is often a requirement to be true to the underlying physics and mathematics; doing otherwise, perhaps by implementing approximations, calls into question the scientific validity of the simulation results. In numerical linear algebra, only a handful of algorithms may exist for a particular task, and inventing new ones is as least as hard as implementing current ones well. Lower bounds on operation count and data movement further indicate when the algorithm space is no longer profitable to explore for the purposes of minimizing end-to-end performance.

Further complicating the design challenge is the fact that time to solution scales with the *product* of the operation count and the time per operation (the inverse of efficiency). Thus, algorithm and implementation cannot be optimized in isolation. This presents a trade-off between the two with interesting consequences. First, unsurprisingly, it may be advisable to choose a lower-cost algorithm that can only achieve modest efficiency. Second, more surprisingly, it may be advisable to choose a costlier algorithm if it can be implemented efficiently. This effect can be seen in algorithms that trade communication for redundant computation. It can also been seen in our previous work on matrix-driven subdivision surface evaluation [22], which trades irregular access to small amounts of data for regular access to large amounts.

We are presented with the grand challenge of minimizing time to solution for programming activities in the field of computational image reconstruction. As HPC practitioners, it is tempting to dive in and start optimizing code. The economic view suggests otherwise—we should dedicate our limited resources to the activity that yields the highest marginal returns on our effort. Considering the broader application at hand, we find that we're in the middle of a stack of technologies that each could yield a benefit. An example of the setting for an MRI application might be:

(I envision a graphic here that has the shape of an onion—each layer is a technology that is built on others, and each layer presents an opportunity to improve the technologies it supports.)

- (Outermost layer)
- ...
- Medical diagnostics — is imaging the best diagnostic tool?
- Medical imaging — Is MRI the best imaging modality in this case?
- MRI Exam — What pulse sequence will best reveal the problem?
- Reconstruction — What's the best way to formulate the reconstruction problem?
- Mathematical optimization — What's the best optimization algorithm to use?
- System matrix — What's the best way to represent and evaluate the system matrix?
- FFT library — What's the best way to do an FFT?
- FMA instruction — What's the best way to do a fused multiply-add operation in hardware?
- ...
- (Innermost layer)

Research is ongoing at all levels of this hierarchy and the state-of-the-art is a moving target. Yet, we believed that iterative formulations of image reconstruction problems have matured to the point where performance optimization effort is justified for their implementations. While iterative formulations are relatively new, their theoretical foundations are well-established, they seem to have broad application and promising results have been obtained in practice. Since the technology to implement them efficiently is nascent, we choose to focus our efforts on performance optimizations for this class of problems.

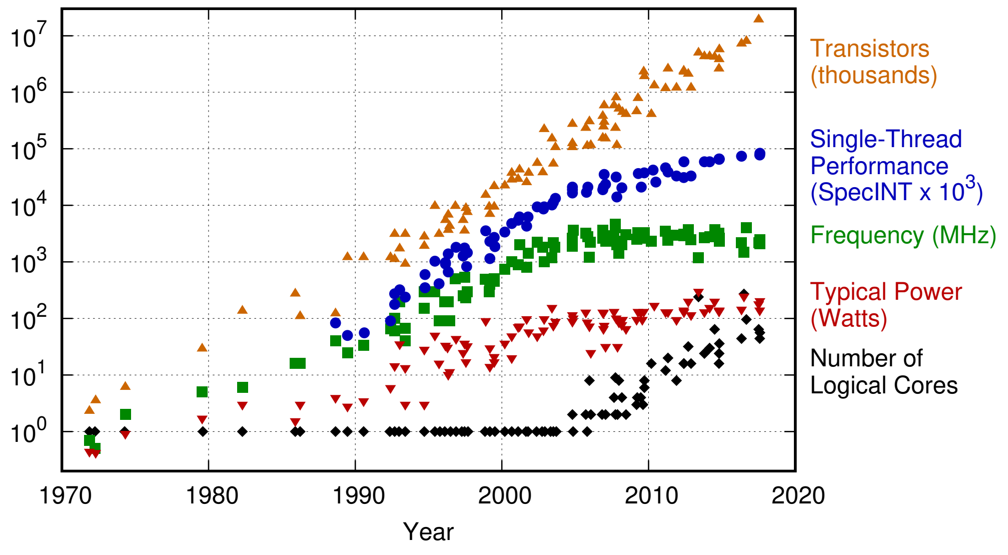


Figure 1.1: Architectural trends in chip density, performance, and energy. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New plot and data collected for 2010-2017 by K. Rupp.

Trends in Computer Architecture

Trends in computer architecture also drive the need for performance portable image reconstructions codes, given the diversity of hardware platforms now used for performance sensitive applications. Figure 1.1 shows the evolution of single-thread computational performance and associated metrics over the last fifty years. The graph features the ongoing march of Moore's Law—the doubling of transistors every eighteen months—but also a prominent inflection point in the other curves in the late 2010s. At that point in time, processor clock frequency leveled off in the low gigahertz range as power reached its practical limit; with the stall in frequency, single-thread performance also languished. To continue delivering performance, computer architects turned to parallelism via multiple processing cores and vector units. Today, modern processors can have tens to hundreds of cores and vector lanes wide enough for sixteen 32-bit floating point numbers.

To attain peak performance on modern processors, it is necessary to write parallel code that leverages multi-core and SIMD architectures. The performance of such codes can be expected to continue improving as core counts increase, provided that the code contains sufficient parallelism to keep the arithmetic units busy. Current estimates show more than a 50% annual increase in floating point operations per unit time [31].

Although computational imaging codes possess sufficient parallelism to leverage multi- and many-core systems, they are typically limited by memory bandwidth rather than peak floating point performance. Memory bandwidth has grown at an average of 23% per year [31], rather than the 50+% computational throughout regime. Furthermore, the gap between the regimes is widening, suggesting that even if a code is compute-bound today, it may be memory-bound in the near future as processing speeds out-pace memory speeds.

Since iterative image reconstruction codes are typically memory-bandwidth bound, and memory bandwidth is trailing advancements in computational throughput, imaging codes are likely to benefit from performance engineering efforts. The most straightforward objective is the development of efficient implementations that run at or near their theoretical limit (as defined by the Roofline model, described later in Section 2). Advanced efforts might also consider algorithmic changes that leverage peculiarities of the underlying platform.

Lastly, the emergence of parallel computing has created a variety of parallel architectures, including multi-core CPUs, vector accelerators, many-core CPUs with fast interprocessor interconnects, and GPUs. Each architecture offers trade-offs in terms of performance, cost, availability, and ease of use. Consequently, imaging practitioners gravitate to different platforms that match their particular requirements. This places additional burdens on imaging software to obtain good performance across a variety of platforms, and to be adaptable to future platforms.

Design Challenges and Goals

The medical and scientific demand for high-performance image reconstruction, along with the gloomy prospect of a memory-bound future, suggests the development of a strategy to deliver much-needed computational performance. This thesis seeks to design a software system that maximizes the *utility* of computation for imaging—the benefit derived from employing computation as a means of discovery. This objective incorporates not just the high-performance computing aspect of system design, but also the human factors that make software systems usable in practice. We believe a system should be:

Productive The system should minimize the time to solution.

Performant The system should fully utilize available processing power.

Portable The system should be performant across a variety of processors, and generations of the same processor.

Scalable The system should be able to solve large problems without additional programming effort.

Expressive The system should provide building blocks for solving a variety of problems in computational imaging.

Collaborative The system should encourage dialogue between application scientists and performance programmers.

Performance Transparent The system should provide feedback to the programmer as to where time is spent during a program execution.

Extensible The system should have a path for incorporating new concepts in the application domain.

Robust The system should provide slow, but correct, code paths when high-performance ones are not available.

Performance and productivity are often at odds, but we will demonstrate a programming system that bridges these competing goals. Our approach allows programmers to express algorithms using standard mathematical building blocks, and our implementation automatically applies major performance optimizations to produce efficient programs. It does this by leveraging a specialized domain—iterative image reconstruction—without claiming to provide a solution to all programming tasks, or even all imaging problems.

Contributions

To fulfill the goals stated above, this thesis makes the following contributions:

- A novel, domain-specific programming language named Indigo for expressing linear operators found in iterative computational image reconstruction.
- A broad set of program transformations that alter programs' performance characteristics while preserving their semantics.
- Case studies of image reconstruction tasks in Magnetic Resonance Imaging (MRI), Message Passing Interface (MPI), ptychography, and microscopy. Studies include example operators and performance results from modern multi-core CPUs, many-core CPUs, and GPUs. Performance results indicate that Indigo operators can achieve up to 90% of the theoretical peak performance, and produce operators that are sufficiently fast to meet clinical and scientific deadlines.
- A methodology for scaling reconstruction to distributed-memory supercomputers, and performance results indicating perfect scaling on a large MRI problem instance.
- An implementation of Indigo available for download at `https://mbdriscoll.github.io/indigo/`.

Thesis Outline

This thesis is organized as follows:

- Chapter 1 introduces the challenge of developing portable, productive, and efficient computational image reconstruction codes. It argues that performance optimization can yield substantial benefits to modern imaging workflows, and the benefits are likely to grow as the performance of memory-bound imaging codes lags processor performance.

- Chapter 2 presents the abstract formulation of iterative computational image reconstruction. It also presents a concrete MRI reconstruction task that serves as a motivating example for the remainder of the thesis. Lastly, it presents the performance analysis framework, the Roofline Model, by which we assess the performance of our eventually reconstruction codes.

- Chapter 3 presents our proposed solution to the task of maximizing utility of computation for image reconstruction—a domain-specific programming language called Indigo. It discusses the structure of an Indigo program and the imaging notions it can express.

- Chapter 4 describes the implementation of the Indigo domain-specific language (DSL). It highlights the software architecture that gives Indigo broad portability, and it lists our techniques for efficiently mapping linear operators to high-performance platforms.

- Chapter 5 catalogs the set of transformations that can be applied to Indigo programs to improve their performance. The existence of these transformations elevates Indigo from a library which blindly evaluates users' programs to a domain-specific language and compiler which broadly re-organize user programs.

- Chapter 6 applies Indigo to a number of reconstruction tasks in computational imaging—MRI, MPI, ptychography, and microscopy. For each, it illustrates the linear operator and presents its performance across high-performance platforms.

- Chapter 7 scales Indigo to distributed-memory supercomputers to solve a large MRI reconstruction task. It gives perfect scaling results that reveal equitable trade-offs between reconstruction time and resources utilized.

- Chapter 8 surveys related work in high-performance computing, computational image reconstruction, and domain-specific language design.

- Chapter 9 concludes by review the design goals from Chapter 1 and scoring the proposed solution. It offers suggestions for future directions of research in achieving widespread and productive performance.

Chapter 2

Background

The development of a productive, high-performance approach to image reconstruction requires familiarity with both the application domain and the underlying programming systems and techniques. This chapter surveys both to provide the background required to understand our approach. First, we define the image reconstruction problem for an abstract linear imaging system. We discuss iterative formulations of reconstruction, including the concepts of a system matrix that models the imaging system and iterative algorithms that converge to a solution that is consistent with acquired imaging data. In the latter half of this chapter, we focus on the implementation techniques that underpin our work. We introduce the concept of a domain-specific programming language and discuss its representation in a computer. Finally, we present the techniques by which we assess the performance of our eventual reconstruction codes.

2.1 Computational Imaging

Imaging is the acquisition of spatial signals, and computational imaging is a special case of imaging in which the acquired signal must be transformed ("computed on" or "reconstructed") to form the desired image. When the imaging system and reconstruction algorithm are designed together, they can make up for for shortcomings in the imaging hardware, provide non-obvious advantages like super-resolution, or offer interesting trade-offs between scan time, resolution, and signal-to-noise ratio. Substantial work continues on hardware-algorithm co-design, hence a system to implement new algorithms is increasingly valuable.

A general representation of a imaging system is a function $f(\cdot)$ that operates a continuous-valued field $\hat{x}$, the intrinsic image, is combined with an additive noise $\hat{n}$, and yields another field $\hat{y}$, the acquired representation of the image:

$$\hat{y} = f(\hat{x}) + \hat{n}.$$

The image reconstruction problem seeks to recover $\hat{x}$ given acquired data $\hat{y}$, i.e. the inverse problem:

$$\hat{x} = f^{-1}(\hat{y}).$$

The operator f is often called the “forward operator” because it represents the forward transformation of the intrinsic image to the acquired representation. It is typically unknowable because of non-idealities in the imaging systems like physical aberrations in a lens. However, it can be modeled well enough to produce images of sufficient quality in many important cases.

System Matrix Formulation

The continuous form of the imaging relationship is impossible to represent in a digital computer, hence we seek a more amenable formulation. We represent the intrinsic and acquired signals as vectors—both discretized and finite in spatial support. Similarly, it is well known that the forward operator is a linear operator in many important modalities, in which cases the forward operator f can be represented by a matrix A. This gives rise to the matrix formulation of the imaging relationship

$$y = Ax + n$$

where x, y, and n are vectors and A is a *system matrix* representing the select behaviors of the continuous linear transformation f. In this formulation, assuming zero noise, image reconstruction takes the form of a matrix inversion problem:

$$x = A^{-1}y.$$

However, there is no guarantee that A is invertible (or even square). Even if inversion were mathematically possible, it is expensive both in time, requiring $O(n^3)$ operations for a matrix with side length n, and space, requiring $O(n^2)$ space.

Mathematical Optimization Methods

A common means of solving an inverse problem is formulating it as an optimization problem in which we search for the best image x that, when transformed by our model A, most closely approximates the acquired data. This can be solved in a least-squares sense by solving the system:

$$A^H y = A^H A x.$$

where A^H is the conjugate transpose or “adjoint” of the matrix A. The matrix $A^H A$ is square, positive semi-definite matrix, and termed the “normal operator”. Thus, we search for a solution to:

$$\arg\min_x \|A^H A x - y\|.$$

We can solve this by defining a new system of equations, $y' = A'x'$, where $y' = A^H y$, $x' = x$, and $A' = A^H A$, and applying the conjugate gradient method.

However, because the original system A is ill-posed, the conjugate gradient answer x' can take on a number of solutions that are consistent with y' but undesirable from an application perspective. For example, an MRI scan for proton density should measure only non-negative values, since for practical purposes there cannot be negative amounts of matter in a volume. The conjugate gradient method, applied to such data, can potentially return negative values in x'.

Because multiple solutions can be consistent with acquired data, it is desirable to choose solutions with known-good properties in a process termed *regularization*. In the proton density example above, a *non-negativity* regularization can be applied to bias the solution toward images that satisfy physical laws. Other examples of regularizations include:

- Wavelet regularization [13] utilizes the observation that most natural images are sparse in the wavelet domain. Images undergoing wavelet regularization first are converted to the wavelet domain via a discrete wavelet transform, thresholded according to a user-defined parameter, and then converted back to the image domain via an inverse transform.

- Total Variation regularization (TV) [9] utilizes the observation that natural images have larges areas with similar values, and thus the variation between adjacent values in small.

- Low-Rank regularization (LR) [62] is commonly performed over a time-sequence of images and utilizes the observation that motion in the images should be represented by only a few parameters. LR regularization requires a singular value decomposition of a matrix formed from time-images, a thresholding step to limit the magnitude of the singular values, and a matrix-matrix multiplication step to reconstruction the now-regularized image.

- Locally Low-Rank regularization (LLR) [62] is similar to low-rank regularization, but applies the low-rank constraint in small blocks of the image, rather than to the entire image.

If performing a regularized reconstruction, one must use an optimization routine that admits regularization. We employ the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) method [5], which is a variant of the Iterative Shrinkage-Thresholding Algorithm (ISTA) method [19] with a modified step size calculation. To formulate a reconstruction problem with system matrix A, acquired data y, and candidate image x, we define a routine for computing the current gradient via the system matrix:

$$G(x) = A^H(Ax - y)$$

and another routine for applying the regularization:

$$R(x) = \lambda \cdot f(x)$$

where λ is a parameter that trades off between data consistency and regularization.

Additional methods for mathematical optimization are likely to benefit from our work. Another common method, Alternating Direction Method of Multipliers (ADMM) [11], has been used for MRI reconstruction CITE. From a performance perspective, iterative methods for mathematical optimization have similar structure—a dominant matrix-vector multiply with an operator derived from the system matrix, and vector-vector arithmetic for computing gradients, applying steps, and checking termination conditions. Because of the similarities shared by common schemes, we don't specialize our techniques for particular ones.

2.2 A Motivating Example: Non-Cartesian SENSE MRI

In subsequent chapters, we propose a new domain-specific language for image reconstruction problems. To make the discussion concrete, we introduce in this section a common reconstruction operator—the SENSitivity Encoding (SENSE) operator for MRI reconstruction tasks. We describe the basics of MRI and the structure of the SENSE operator. This operator constitutes one the of several case studies we present later in Chapter 6.

MRI Imaging Setup

MRI is a versatile imaging modality that can be used in a variety of ways. For simplicity, we focus on a straightforward scheme that acquires an image representing the density of protons within a volume of interest. To perform the scan, the volume of interest is positioned with the scanner. The scanner then plays a "pulse sequence" of radio waves that excite protons within the volume. Additional magnetic fields induce spatially-varying, resonant spinning of the protons, effectively encoding their position as frequency and aggregate density as amplitude. One or more inductive coils arranged around the volume of interest record the emitted signal in separate receive channels, and a Fourier transform of the channel data yields spatially-weighted variants of the intrinsic image. The use of multiple inductive coils to receive signal simultaneously is termed "parallel imaging" (PI). PI enables shorter scan times, but requires a post-scan computational procedure to combine coils' images into one. More background on MRI image formation can be found in [51].

MRI scans employ a "scan trajectory" that dictates how the volumetric signal is sampled. Scan trajectories can sample arbitrary signal frequencies, but physical limitations, sampling theory, and application demands have resulted in a few common ones. Cartesian scans sample at regularly-spaces intervals in the sampling domain, and their output can be represented in a dense, multi-dimensional array of sample data. Other scans cover the sampling space via patterns as spirals, circles, and spokes. Since they sample at non-Cartesian points, they must use a sparse representation for their output. For our purposes, the acquired signal in a non-Cartesian scan can be represented as a list of $(\vec{x}, z)$ pairs in which $\vec{x}$ represents the coordinates of the sample and z represents the sample as a complex number.

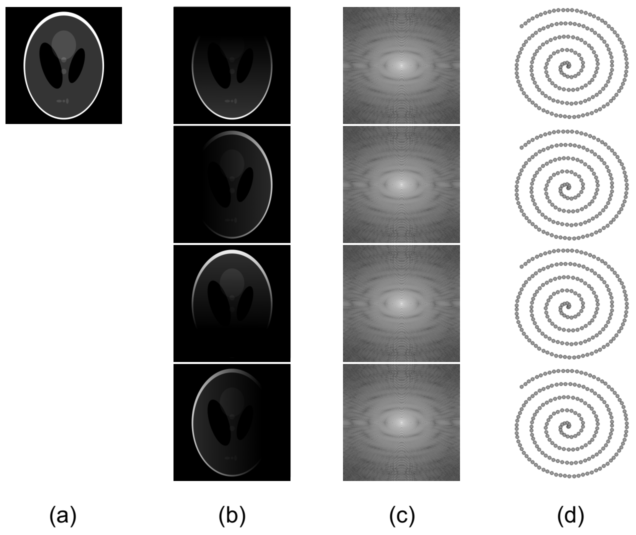


Figure 2.1: The non-Cartesian SENSE operator transforms a proposed image (a) into sparse data acquired by the scanner, represented here by a spiral trajectory (d). Along the way, the proposed image is spatially weighted to produce “channel images” (b), and the channel images undergo a centered discrete Fourier Transform (c) before being re-sampled along the scan trajectory.

SENSE Forward Operator

The SENSE *forward operator* models the transformation from the proposed reconstructed image to the acquired scan data. For non-Cartesian trajectories, the image is represented by a dense, multi-dimensional array; the scan data is represented by the list of location-value pairs described above.

The SENSE forward operator models the transformation from a proposed MR image into acquired scan data in two major steps, as shown in Figure 2.1. First, it multiplies the image by a number of “sensitivity maps,” each representing the spatial sensitivity of the corresponding receive coil. The intermediate “coil images” each represent the view of the volume of interest from the perspective of a coil and account for signal attenuation due to the inverse square law. These coil images then each undergo a discrete, centered non-uniform Fast Fourier Transform (NUFFT) [23]. Finally, the signal values are scaled by a density compensation factor representing the inverse density of samples in a local region of the image, and ensuring that the signal has uniform density across the image. The density-compensated result is the acquired data according to the SENSE model. Mathematically, the relationship between the acquired image k_i and the intrinsic image ρ can be expressed as

$$k_i = P\mathcal{F}S_i \cdot \rho, \quad \forall i \in c \tag{2.1}$$

where P represents the density compensation factor, $\mathcal{F}$ represents the NUFFT and S_i represents the sensitivity map for the i-th receive channel. If P and S_i were represented as matrices, they would be diagonal.

The centered NUFFT can be defined as the product of several linear transformations:

$$\mathcal{F} = GCFCA.$$

In particular, these operators are:

- G is a gridding matrix that transforms the signal from a Cartesian representation to a non-Cartesian representation. It does this by blurring the signal and re-sampling it at the desired non-Cartesian coordinates. The width of the blur kernel has important implications for accuracy, memory footprint, and running time of the gridding operation [4].
- C is a FFT-shift operator. FFT shifts can be implemented by swapping opposite quadrants of an two-dimensional image (or opposite orthants of a three-dimensional image), or by modulating the image with a high-frequency sinusoid to induce the shift in the corresponding Fourier domain.
- F is a standard Fast Fourier Transform (FFT).
- A is a "pre-apodization" operator the performs an elementwise multiplication in anticipation of the subsequent gridding operator. Since the gridding operator blurs the image in the frequency domain, the pre-apodization operator serves to negate this effect via multiplication in the spatial domain.

2.3 Domain-Specific Languages

Domain-specific programming languages are languages designed only to solve problems in a particular domain. As such, they tend to have substantial expressiveness within their target domain, but limited expressiveness beyond it. This arrangement can reduce the effort required to implement programs, and hence reduce the overall time to solution for domain tasks.

Programs written in domain-specific languages can be represented and manipulated much in the same way the general-purpose programming languages are. To execute a program, it must typically undergo three steps:

- Parsing transforms program text into an intermediate representation, commonly an abstract syntax tree (AST). The AST is a data structure representing the structure of the program independently of its input format. They are typically represented as trees in which leaf nodes represent numbers, variable names, and character strings; similarly, interior nodes represent mathematical operations between child trees or program entities like classes, functions, function calls, and control flow logic like loops.

- The AST then undergoes a series of transformations resulting in a new AST. The new AST represents a different program, but it must compute the same result as the original program. If it does, the transformations are said to preserve the semantics of the program. The example AST above might be transformed into the new statement $z = A \cdot (x + y)$ using the distributive property of matrix multiplication over addition. In general purpose compilers, these transformations include loop blocking, loop fusion, loop interchange, code motion, dead code elimination, and common subexpression elimination [1].

- Finally, the AST is converted into machine-executable code via a *code generation* step. Here, machine code is generated for each function, arranged in a memory image, and translated into an executable binary.

The three-step parse, transform, and code generate process isn't fundamental to implementing programs, however. Interpreted language implementations omit the code generation step, instead traversing and "executing" ASTs.

Some languages omit the parsing step and instead construct ASTs via other means. "Embedded" DSLs are languages that share another language's syntax. This arrangement allows them to share the other languages parsing facilities as a means of constructing ASTs. ASTs can also be constructed directly because they themselves are data objects that a program can manipulate.

2.4 Performance Profiling

In developing high performance software, it is desirable to measure the performance of a code section and to calculate its computational efficiency. These data can then be used to inform future decisions about performance optimization effort. When working with a domain-specific language, an additional opportunity arises to incorporate profiling support into the language to provide rich information associated with program entities.

We employ the Roofline model of performance [67] to assess the efficiency of our implementations. The Roofline model seeks to bound the performance of a kernel as a function of the performance characteristics of the underlying platform and the *arithmetic intensity* of said kernel. It defines arithmetic intensity as the ratio between the total number of floating point operations the kernel performs and the total number of memory words transferred across a link between fast and slow memory, typically the last-level cache and main memory.

Under this formulation, a kernel's arithmetic intensity is both a function of the algorithm it employs and the characteristics of the platform on which it runs. Consider the following examples:

- Computing the vector sum of two n-length vectors requires computing n flops, reading $2n$ words from memory, and writing n words to memory, for a arithmetic intensity of

$$a_1 = n/3n = 0.33.$$

This is among the lowest arithmetic intensities and signifies a highly memory-bound kernel.

- Computing a dense matrix-matrix product of square matrices with dimension n requires $2n^3$ flops and one read and write of the matrices ($6n^2$ words), for an arithmetic intensity of $a_2 = n/3$. For typical matrix sizes, this is among the highest arithmetic intensities and signifies a highly compute-bound kernel.

- Computing the FFT of a n-length vector requires $5n \cdot \log n$ flops and $2n$ words, for an arithmetic intensity of:
$$a_3 = \frac{5n \log n}{2n} = \frac{5}{2} \log n.$$

- Computing the three-dimensional FFT of a array with side length n can be evaluated as one-dimensional FFTs along "pencils" in each of the three dimensions. There are a total of n^2 pencils in each dimension, so the total number of flops is $3n^2 * 5n \log n$. Calculating the data movement cost is more complex because data re-use opportunities arise. Now, we must consider the size of the fast memory:

  1. If the fast memory is large enough to hold the entire three-dimensional array, then it must only be read and written once, for a total of $2n^3$ words and an arithmetic intensity of
  $$a_4 = \frac{3n^2 \cdot 5n \log n}{2n^3} = 7.5 \log n.$$
  2. If the fast memory can only hold one pencil, then the entire three-dimensional array must be transfered six times—a read and write for each dimension—yielding an arithmetic intensity of:
  $$a_5 = \frac{3n^2 \cdot 5n \log n}{6n^3} = 1.67 \log n.$$
  3. If a two-dimensional "slab" of the three-dimensional array fits in fast memory, then the array need only be transferred four times—a read and write for the first two one-dimensional FFTs, and a read and write for the third dimension FFT. This results in an arithmetic intensity of:
  $$a_6 = \frac{3n^2 \cdot 5n \log n}{4n^3} = 2.5 \log n.$$

  This latter most case is common for image reconstruction problems ($n \approx 512$) running on multi-core processors and GPUs with cache sizes in the 512 KB to 6 MB range.

We then use the arithmetic intensity of a kernel to determine its peak attainable flop rate on a particular platform. The Roofline model asserts that a kernel is ultimately limited either the peak floating point performance of the machine, or by the ability of the memory system to deliver data to the processor (i.e. the peak sustained memory bandwidth). The full Roofline model identifies a series of performance bounds that are attainable with increasingly complex code optimization, but the simplified version is sufficient to analyze our codes. We measure these two machine properties via micro-benchmarks—for peak flop rate, we compute the flop rate of a vendor-optimized general matrix-matrix multiplication (GEMM) routine on large, square matrices; for peak memory bandwidth, we use the STREAM benchmark [46].

These two properties give rise to the eponymous Roofline plot that relates arithmetic intensity to peak attainable floating point performance. Figure 2.2 shows the Rooflines of the multi-core CPU, many-core CPU, and GPU platforms we evaluate. In each case, the horizontal component of the Roofline curve represents the peak floating point performance of the machine, and the slanted component represents the penalty incurred for low-intensity kernels. The slope of the slanted line is exactly the STREAM bandwidth number. The intersection of the two lines represents the balance point at which the memory system delivers just enough data to keep the arithmetic units busy.

Given a Roofline plot for a machine of interest, we can locate a kernel's arithmetic intensity and read off the corresponding peak attainable performance. This "Roofline peak performance" represents an upper bound on the performance attainable for the given architecture and algorithm. By measuring the performance of a kernel, we can assess how close it is to this optimal rate, and conversely how much performance remains to be recovered.

Beyond providing bounds on performance, the Roofline model also guides performance optimization efforts. Depending on a kernel's arithmetic intensity, it may be worthwhile pursuing a variety of optimizations. Memory-bound kernels benefit from software pre-fetching and awareness of non-uniform memory access (NUMA) domains. Compute-bound kernels benefit from thread-level and data-level parallelism, and instruction mixes with balanced numbers of additions and multiplications.

It is important to be precise about the assertions that can be drawn from a Roofline analysis. Its chief utility is assessing how well a algorithm implementation leverages the fast memory in a particular platform. Attaining peak Roofline performance does not imply that the algorithm in use is the best algorithm for solving the problem at hand; it merely asserts that the chosen algorithm is running efficiently. For example, it is possible to achieve peak performance for a classical matrix-multiplication implementation ($O(n^3)$ flops), but using Strassen's $O(n^2.80)$ algorithm [59] might achieve a smaller fraction of peak but yield better overall run times. In this thesis, we use the Roofline model the demonstrate that our chosen algorithms make efficient use of the underlying hardware, and we defer to expert opinion to reason about their overall efficiency.

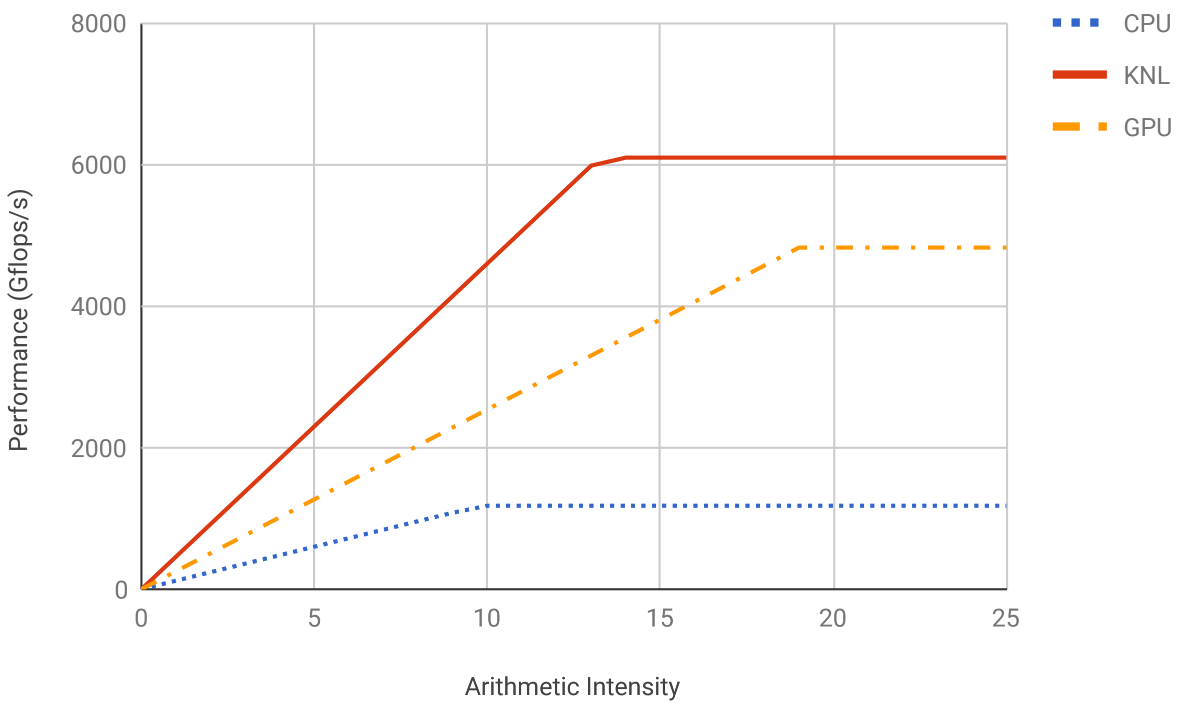


Figure 2.2: Roofline plots of the three platforms we evaluate: an Intel Xeon E5-2698 CPU in the Cori supercomputer, an Intel Xeon Phi 7250 ”Knight’s Landing” (KNL) many-core processor also in Cori, and a Nvidia Kepler K20X GPU on a private system. The slanted portion of the Roofline has a slope corresponding to the peak memory bandwidth attained by the STREAM benchmark. It represents the region in which kernels with low arithmetic intensity are memory-bound. Similarly, the flat region represents the peak FLOP rate of the machine, and it spans the region of arithmetic intensities in which kernels are compute-bound.

Chapter 3

Language Design

In this chapter, we propose a domain-specific programming language named Indigo for implementing image reconstruction codes. We survey the challenges that arise in implementations of such programs and discuss the competing concerns of productivity and performance that arise in this context. The Indigo language balances these concerns, and we argue that it strikes a good balance between productivity and performance. We conclude with limitations of the domain-specific language approach, as well as directions for future work.

3.1 Design Challenges

One goal of this thesis is to develop techniques for achieving peak computational performance in implementations of computational image reconstruction codes. This goal can be refined in the context of the two major interested parties—the domain scientist and the high-performance computing expert.

The domain scientist is focused on his or her science mission, and uses software development as a tool for advancing it. He or she is not concerned about computational efficiency, except when it interferes with scientific progress. For the domain scientist, we seek a representation of programs that delivers good performance on average without substantial effort, and that uses concepts natural to the application domain.

The high-performance computing expert aims to develop codes that take maximum advantage of the platforms on which they run. He or she wants to explore many different implementations of the program that offer potential advantages for performance. For the HPC expert, we seek a tool that enables him or her to rapid apply and assess different performance optimization strategies. The tool should also provide rich feedback about performance characteristics of the program that it runs. The ability to assess many implementation variants can lead to implementations that surpass expert-authored implementations in standard, general-purpose languages simply because the latter inhibits rapid iteration for performance optimization [56].

Additional opportunities arise when the domain scientist and HPC expert collaborate.

Here, the domain scientist might author a correct, but slow, program that he or she can transfer to a HPC expert to optimize. We seek to facilitate this conversation by choosing a level of abstraction that is familiar to both parties. The HPC expert might also seek to automate some of the optimizations they implement, thus allowing domain scientists to apply them automatically. Automation of performance optimizations multiplies the impact of the HPC expert since it encodes their expertise in a way that scales beyond a handful of users.

3.2 The Linear Operator Abstraction

Many computational imaging problems involve linear transformations of the signal, so the Indigo language is designed around the concept of a linear operator multiplying an operand matrix. This level of abstraction is sufficiently general to capture much of computational imaging, but sufficiently restrictive that it can be analyzed and transformed to meet performance goals.

Mathematically, a linear operator $\tilde{A}(\cdot)$ is defined as an operator that satisfies

$$\tilde{A}(x+z) = \tilde{A}(x) + \tilde{A}(z)$$

for any $x, z \in \mathbb{C}^n$. The domain of $\tilde{A}$ is $\mathbb{C}^n$, and the range of $\tilde{A}$ is $\mathbb{C}^m$ in general. Every linear operator can be expressed as a matrix, so we can rewrite the previous relationship in matrix notation for the corresponding matrix A:

$$A(x+y) = Ax + Az.$$

Here, A is a matrix with m rows and n columns.

Indigo provides an abstract notion of a linear operator that can be implemented by various different operator classes. Here, we catalog the commonalities of all operators in Indigo. In an object-oriented sense, they can be divided into two groups: instance properties that describe a given operator and methods that implement evaluation functionality.

Operator Properties

Operators in Indigo provide a number of properties that describe their underlying characteristics. First are mathematical properties that describe the shape of the equivalent operator matrix A:

A.m : dimensions of an output vector, or the number of rows of the operator matrix.

A.n : dimensions of an input vector, or the number of columns of the operator matrix.

Second, the operator must provide implementation details that are of interest to analysis and transformation tasks. These include:

A.datatype	The underlying datatype of A. Operator and operand datatypes must match for an evaluation to proceed.
A.storage_size	The amount of memory in bytes required to store this operator.
A.scratch_size(k)	The amount of scratch memory in bytes required to evaluate this operator against a matrix with k columns.
A.num_flops(k)	The minimum number of floating point operations required to multiply this operator against a matrix of k columns.
A.num_bytes(k)	The minimum number of bytes expected to travel between last-level cache and main memory when this operator is multiplied against a matrix of k columns.

These properties are optional to provide, but enhance feedback to both users and optimization techniques.

Operator Evaluation Routines

Each operator object must also provide functionality to evaluate itself against vectors and matrices. Since vectors can be interpreted as matrices with one column, we only consider the case of matrix-matrix products. At any particular evaluation, the operator can be interpreted in normal, transposed, or conjugate-transposed orientation; likewise, it can be located on the left-hand side of the operand or on the right-hand side. This leads to six variants of matrix-matrix multiplication:

$Y = AX$	normal left-hand multiplication	$Y = XA$	normal orientation, right-hand multiplication
$Y = A^TX$	transposed left-hand multiplication	$Y = XA^T$	transposed orientation, right-hand multiplication
$Y = A^HX$	conjugate-transposed left-hand multiplication	$Y = XA^H$	conjugate-transposed orientation, right-hand multiplication

Operator Classes

Indigo provides implementations of many common operators useful in implementing reconstruction codes. Here, we survey these operators in anticipation of introducing a grammar for Indigo which defines how operators can be combined in useful ways.

The most basic operator is the Matrix. It is sufficiently general to represent any linear transformation. However, many linear transformations admit faster evaluation algorithms, or more space-efficient representations, than the Matrix operator. When desirable, Indigo implements specialized "matrix-free" operators to enhance performance. They include multiplication by an identity matrix or a matrix of ones; element-wise multiplication; and a

```
Operator : LeafOp    | UnaryOp
         | BinOp     | NaryOp ;

  LeafOp : Identity | FFT
         | Matrix   | OneMatrix ;

 UnaryOp : ‘Adjoint’ ‘(’ Operator ‘)’
         |   ‘KronI’ ‘(’ Operator ‘)’
         |   ‘Scale’ ‘(’ Operator ‘)’ ;

   BinOp : ‘(’ Operator ‘+’ Operator ‘)’
         | ‘(’ Operator ‘×’ Operator ‘)’ ;

  NaryOp : ‘BlockDiag’ ‘(’ Operator+ ‘)’
         |    ‘VStack’ ‘(’ Operator+ ‘)’
         |    ‘HStack’ ‘(’ Operator+ ‘)’ ;
```

Figure 3.1: The Indigo grammar comprises leaf operators, which represent atomic linear transformations, and compositional operators, which combine other operators structurally or mathematically.

multi-dimensional Fast Fourier Transform. More operations such as the discrete wavelet transforms are certainly possible but unimplemented. These operators serve as terminals in the operator grammar.

To build complete operators found in actual reconstruction codes, the aforementioned operators are typically combined structurally or algebraically. Operators can be multiplied or added with other operators (dimensions permitting); stacked vertically, diagonally, or horizontally; transposed or conjugate-transposed; scaled; and Kronecker multiplied. Rather than performing these operations immediately during operation construction, Indigo captures these relationships in an abstract syntax tree. This representation enables tree transformations that dramatically improve operator performance.

The space of Indigo programs can be described by the grammar in Figure 3.1. Here, the basic Matrix and matrix-free operators as classified as `LeafOps` and the structural and algebraic operators are `UnaryOps`, `BinOps`, and `NaryOps`.

3.3 The Indigo Language Definition

Among the many challenges of creating a new programming language is defining the interface by which the programmer authors a new program. Traditionally, this has been accomplished via a new language syntax and associated lexical analyzer and parser for converting program text into an abstract syntax tree (AST). The traditional approach is expressive because the language syntax can closely model the abstract program model, but it has some practical drawbacks. First, implementing and maintaining a consistent and correct lexical analyzer and parser is a non-trivial, error-prone task. Second, programs written in the new language are typically inoperable with program written in other languages (unless great care is taken to ensure otherwise) and thus don't benefit from the ecosystem of libraries and development tools surrounding established languages. As a net result, new languages often must re-invent the wheel before reaching useful status.

Indigo takes a different approach—the domain specific embedded language (DSEL). Instead of implementing a custom language syntax and compiler frontend, Indigo programs are constructed directly in the Python programming language. To build an application-level Indigo operator, the user instantiates Operators and combines them via structural and algebraic operators to form an Indigo abstract syntax tree (AST). In this approach, the user builds the Indigo AST directly in memory using Python, rather than relying on a compiler frontend to convert program text into an AST. The embedded approach allows users to leverage the tools of the Python ecosystem in building and running Indigo programs. We routinely used the Python debugger and profiler in development, and our Python+Indigo applications make extensive use of popular libraries for foreign function interfaces, multi-dimensional arrays, and input-output routines.

The embedded domain-specific language is not without its drawbacks. First and foremost, embedding one language in another can cause friction when the host language cannot directly express an semantic notion from the embedded language. For example, in Indigo it is desirable to give names to intermediate operators for visualization purposes. One might construct the following expression using the Python syntax for multiplication:

```
nufft = grid * cfft * apod
```

Upon evaluation, `nufft` will be an Operator as desired; however, it lacks a useful name and setting one must be done in a separate statement:

```
nufft = grid * cfft * apod
nufft.name = "nufft"
```

Alternatively, Indigo users can construct the same operator via Operator constructors:

```
nufft = Multiply([grid, cfft, apod], name="nufft")
```

These approaches have various degrees of verbosity, but they are equivalent and both are provided to users who may prefer the style of one or the other.

Lastly, it is important to note that the decision to embed Indigo in Python does not preclude the development of a custom syntax.

Because of mismatch between Python's syntax and the Indigo , the resulting programming experience is slightly more cumbersome than performing the equivalent task in a purpose-built language, but the user can leverage all the features of Python to aid in AST construction. The Indigo approach to embedded AST construction does not preclude the development of a custom syntax, but such work is not necessary to demonstrate the feasibility of our approach.

An Example of Constructing an Indigo Operator

To make things concrete, we present the procedure for constructing the forward operator for the Magnetic Particle Imaging (MPI) example (described in detail in [38] and Section 6.5). The MPI forward operator, A, is given as:

$$A = SD$$

where S is an operator that selects overlapping patches of the image:

$$S = \begin{bmatrix} I_s & & & & & \cdots \\ & I_r & & & & \\ & & I_s & & & \\ & & I_s & & & \\ & & & I_r & & \\ & & & & I_s & \\ \vdots & & & & I_s & \ddots \end{bmatrix}$$

and D subtracts the average value from each patch:

$$D = \begin{bmatrix} R & & \\ & \ddots & \\ & & R \end{bmatrix}, R = I_p - \tfrac{1}{p}\mathbb{1}.$$

Here, $\mathbb{1}$ is a matrix of ones and s, r, and p are all positive integers. The forward operator A is constructed as follows.

1. Instantiate a matrix of ones of size p-by-p: $T_1 = \mathbb{1}$.
2. Scale by $1/p$: $T_2 = \frac{1}{p} \cdot T_1$.
3. Instantiate an Identity operator of side length p: $T_3 = I_p$.
4. Take the difference to construct R as specified above: $R = T_3 - T_2 = I_p - \frac{1}{p}\mathbb{1}$.
5. Instantiate an Identity operator with one diagonal element for each of c panels: $T_4 = I_c$.
6. Take the Kronecker product to induce replication along the diagonal: $D = I_c \otimes R$.
7. Generate S by creating a sparse matrix with ones in strategic locations. Matrix entry (i, j) should be one if and only if the i-th panel pixel should be copied from the j-th image pixel.

8. Multiply S and D to yield the MPI forward operator: $A = SD$.

The normal operator $A^{\perp}$ can then be computed as the product of the adjoint and forward operators:

8. $A^{\perp} = A^{H} \cdot A$.

The resulting $A^{\perp}$ can be used in a gradient method to compute a solution to the original inverse problem.

Chapter 4

Language Implementation

The Indigo language must balance competing demands for application expressiveness and performance portability. Maximizing both dimensions requires co-design of the language and the underlying runtime. In this chapter, we describe techniques for efficiently mapping Indigo programs onto multi-core CPUs and GPUs.

4.1 Indigo Software Architecture

As a software artifact, the Indigo implementation is made up of three components:

- The Indigo *frontend* presents a collection of linear operators that can be arranged to implement a variety of image reconstruction codes. The operators are platform-agnostic, so Indigo programs possess the characteristic write-once, run-anyway property.
- Indigo *backends* are responsible for evaluating operators on particular platforms. The backend can be selected by the user at runtime from a list of available backends. Backends typically represent execution resources—CPUs and GPUs.
- Finally, the Indigo *runtime* is responsible for scheduling operators on the user-selected backend. It is most similar to an interpreter—it traverses the operator tree and dispatches to the approach backend evaluation routines.

The Indigo Backend Abstraction

An Indigo backend implements the functionality necessary to evaluate the set of operators on a particular hardware platform. To achieve this, backends must implement interfaces for computation and memory management.

Linear Transformation Routines

Indigo operators must provide implementations of:

`cgemm` : complex-valued dense matrix-matrix multiplication

`ccsrmm` : complex-valued CSR format sparse-times-dense matrix multiplication

`cdiamm` : complex-valued diagonal-format sparse-times-dense matrix multiplication

`conemm` : complex-valued matrix of ones times dense matrix multiplication

`fftn` : n-dimensional Fast Fourier Transform

`ifftn` : n-dimensional Inverse Fast Fourier Transform.

These six routines are sufficient to implement high-performance reconstruction codes for a variety of applications.

Vector-Vector Arithmetic Routines

Though not critical to the forward operator implementations, backends can also provide implementations of the following BLAS-1 routines to speed up vector-vector bookkeeping in iterative reconstruction algorithms:

`caxpy` : complex-valued vector-vector scaling and addition

`cdot` : complex-valued dot product

Memory Management Routines

Lastly, backends must implement routines for managing memory. This interface exists to support GPU devices which possess distinct memory spaces for host and device memory.

`malloc` : allocate memory

`free` : free memory

`copy_to_device` : copy data from host to device

`copy_to_host` : copy data from device to host.

Together, this set of routines provides the interface for Indigo's runtime to schedule and evaluate linear operators on both modern CPUs and GPUs.

Existing Indigo Backends

Indigo provides implementations of backends for common platforms.

mkl The MKL backend targets multi-core and many-core CPUs via Intel's Math Kernel Library (MKL).

cuda The CUDA backend targets Nvidia GPUs via the cuBLAS and cuSPARSE libraries.

numpy The Numpy backend targets CPUs via the Numpy library. This backend serves as a "reference backend" that implements slow, but correct and readable, routines for operator evaluation. Furthermore, this backend is useful for debugging because Numpy implements array-bounds checking and type-checking.

customcpu The Custom CPU backend implements customized versions of the `csrmm` and `diamm` routines that leverage the exclusive-write property of certain matrix-matrix products to avoid unnecessary synchronization. For the remainder of the functionality, the Custom CPU backend uses MKL.

customgpu The Custom GPU backend implements customized `csrmm` and `diamm` routines like the Custom CPU backend, but in CUDA code. Similarly, the remainder of the routines are implemented via cuBLAS and cuSPARSE.

4.2 Implementation of Kronecker Product Operators

Most of the user-level operators classes directly map to a computational routine implemented by the Indigo backends. However, one operator—the Kronecker product—has no low-level equivalent. Indigo instead implements Kronecker product operators via multiple matrix-matrix multiplication routines. Recall the matrix relationship for Kronecker products:

$$Y = (B \otimes A) \cdot vec(X) = AXB.$$

When the Indigo runtime evaluates a Kronecker Product operator, it uses two matrix multiplication operations and a temporary matrix T.

$$T = XB$$

$$Y = AT.$$

Deferring to matrix multiplication routines reduces the implementation burden on backend developers, and still preserves reasonably good performance for general Kronecker operators.

In image reconstruction settings, however, it is common that either the A or B matrix is the identity matrix. In this case, opportunities exist to reduce memory bandwidth by skipping one of the evaluation steps. Here, Indigo only computes $Y = XB$ if $A = I$; similarly it only computes $Y = AX$ if $B = I$.

Returning to the case of general A and B, we observe that further performance gains may be had by carefully re-associating the Kronecker evaluation, i.e. computing $Y = (AX)B$ instead of $Y = A(XB)$. Indigo does not implement this functionality, but here we propose a heuristic for making this decision. If matrix A is m-by-n and matrix B is p-by-q, then computing $(AX)B$ requires $O(mnp + mpq)$ flops while computing $A(XB)$ requires $O(npq + mnq)$ flops. The runtime selects the option that reduces overall flops.

4.3 Matrix Representations

The Matrix operator presented in Chapter 3 places no requirements on the in-memory representation of the matrix in the computer. Because the matrices found in image reconstruction problems have vastly different sparsity structure, it is advisable to use matrix storage formats and associated matrix multiplication routines that complement the matrix sparsity structure.

Indigo provides implementations of the following matrix storage formats:

DENSE A dense matrix scheme stores all matrix values, regardless of their value, in a two-dimensional array. It requires $O(mn)$ memory for an m-by-n matrix, and it can represent any matrix.

CSR A compressed sparse row (CSR) scheme stores only non-zero values and their corresponding column indices, sorted and indexed by an array of row pointers. It requires $O(m + 2 * nnz)$ memory to store an m-by-n matrix with nnz non-zeroes, and it can represent any matrix.

DIA A diagonal storage scheme stores matrix diagonals as contiguous vectors, and also maintains a vector of diagonal offsets. It requires $O(d * max(m, n) + d)$ memory to store an m-by-n matrix with d non-zeros diagonals, and it can represent any matrix.

ONE A matrix of ones can be represented merely by its dimensions m and n, and requires no additional storage. It cannot represent arbitrary matrices.

Matrix Storage Format Selection

With a variety of options for representing matrices, the questions arises of how to choose the appropriate storage format, and whether the programmer or the system makes the choice. We identify two likely schemes:

- In an automated scheme, the programmer supplies the matrices in any format, and the system identifies a good storage format and converts the matrices to it. This approach is attractive because it shelters the programmer from the challenge of choosing a good format, but it requires a heuristic engine to identify good candidate formats.

- The programmer explicitly specifies the matrix storage format to be used, and the system utilizes it. This approach makes it possible for expert programmers to control the eventual layout, but requires more expertise than an automated approach.

Indigo employs a hybrid of the two approaches—it expects programmers to provide matrices in a storage format that complements the sparsity structure, but it may change the storage format of transformed matrices during the operator transformation process. It defers to the underlying matrix library (numpy) for decisions about resulting formats. Examples of format transformations include:

- `VStack( DIA, DIA )` → `DIA`
- `Multiply( CSR, DIA )` → `CSR`
- `Multiply( DENSE, CSR )` → `DENSE`

If the default heuristics don't result in a desired format, programmers can write a custom transformation pass to create their intended configuration.

4.4 Performance Profiling Support

After implementing a linear operator, programmers often seek to optimize it's performance. Empirical performance metrics can be gathered through profiling, the process by which the performance characteristics of a program are measured during execution. As a language that aims to enable high-performance linear operators, Indigo provides custom support for profiling in the form of a "profiling mode." When profiling, Indigo operators report their execution time, memory footprint, and lower bounds on their operation count and DRAM memory bandwidth. From these data, a flop rate and observed memory bandwidth can be computed for each operator. Using a priori knowledge about the machine's peak flop and memory bandwidth rates, the profiler computes the fraction of Roofline peak that each operator is achieving. This is reported back to the programmer to inform future transformation and optimization decisions.

Profiling an Ensemble of Operators

Individual operator data can be misleading when profiling an ensemble of operators that is found in non-trivial reconstruction codes. To understand why, consider an example with two operators:

- Operator A runs in 5ms at 20% of the Roofline peak
- Operator B runs in 95ms at 70% of the Roofline peak.

A naive programmer comparing the operators' fraction of Roofline peak might direct additional optimization effort at Operator A because it's running at a much lower fraction of peak performance. However, Operator A represents a small fraction of the overall runtime—less than 5% total—so recovering the additional 5× performance yields less than a 5% overall speedup. Instead, the programmer should focus on Operator B which could yield a 30% overall speedup to the ensemble.

When profiling an ensemble of operators, we can use profiling data to perform a "holistic" performance analysis. Here, we focus on the optimization opportunities for each kernel with respect to the overall ensemble. We can compute a "bounty" for each operator that represents the performance improvement to be gained if the operator is optimized to its peak Roofline performance. The bounties for the above operators are:

- Operator A could run 5× faster, yielding a bounty of 4 ms.
- Operator B could run 30% faster, yielding a bounty of 28.5 ms.

Then, we sum the bounties to determine the overall speedup available: 32.5 ms. Finally, we compute the fraction of available speedup each operator represents:

- Operator A possesses 12.3% of the available speedup.
- Operator B possesses 87.7% of the available speedup.

Clearly, further optimization efforts are most profitable if directed at Operator B.

Profiling Methodology for Operator Transformations

Indigo's profiling features can also inform decisions about the profitability of transformations to the operator tree. The set of transformations is described in detail in Chapter 5, but for purposes of this chapter all transformations can be characterizes as operations replacing some subset of the nodes in the operator tree with another set of nodes. For example, the product of two diagonal matrices can be replaced by a single matrix whose elements are the product of the original matrices' elements. When considering such a transformation, the programmer can incorporate performance data to assess its profitability relative to the overall runtime of the operator. This technique generalizes the per-operator approach given in the previous subsection by treating sets of operators as "super operators" and applying the same analysis.

Given the performance of operators in one tree configuration, it is generally difficult to predict their performance in some transformed configuration. Furthermore, it can be laborious to measure the performance of a transformed configuration empirically by implementing a candidate transformation. In order to guide optimization efforts without the herculean task of implementing every possible transformation, we suggest using *bounds* on performance to estimate the aggregate benefit to the operator. Conservatively, we can assert that transformations cannot do better than an infinite speedup, with a corresponding runtime of 0 ms.

An infinite speedup is equivalent to removing the original subtree from the original operator tree, and subtracting its running time from the operator's aggregate evaluation time. The resulting time can be compared to the original tree's evaluation time, and the difference serves as a upper bound on the profitability of the transformation.

The conservative approach of assuming infinite speedup is frequently not a tight lower bound on the performance; with expert knowledge, we can frequently do better. Consider a simple tree representing the product of two uni-diagonal matrices. An operator fusion transformation can transform the tree into a third operator consisting of a single matrix. The conservative lower bound on the transformation trivially asserts the new operator will run no faster than 0 ms; however, because we know the structure of the matrices involved in the transformation, we can assert that the new operator will run no faster than the maximum speed of either original operator.

Chapter 5

Program Transformations

As describe thus far, the Indigo language provides a set of linear operator classes and an implementation strategy for evaluating them quickly on a variety of high-performance platforms. This set of features alone is more akin to a standard software library than a domain-specific programming language in that it "blindly" evaluates the program as specified by the user. The performance of the resulting program will be derive directly from the application programmer's arrangement of computation, and consequently the burden of implementing fast programs will fall on his or her shoulders.

To achieve peak performance, it is typically necessary to transform the program to complement the structure of the underlying platform. In the context of the Indigo DSL, transformations take the form of changes to the operator tree—replacing nodes or subtrees with equivalent nodes or subtree. Of course, the transformations we apply must not change the meaning of the program, so any proposed transformation must be justified with a proof of correctness. Even when a transformation's effect on correctness is ambiguous, or only correct under certain unknowable conditions, we must disallow it to be conservative. In general purpose programming languages, it is typically difficult to prove the correctness of large-scale code transformations, and thus general-purpose compilers and interpreters often do not implement broad program transformations that improve performance.

By design, Indigo restricts the domain of programming notions to those found in iterative computational image reconstruction. Furthermore, it adopts the intermediation representation of structured linear operators. In this setting, it is dramatically easier to determine the correctness of program transformations. This chapter describes the transformations available to Indigo programs, their anticipated effect on performance, and techniques for deciding and directing which transformations should be applied to a program.

5.1 Transformation Catalog

Transformation in Indigo are applied to the operator abstract syntax tree. Since nodes in the operator tree implement various linear transformations, we can reason about transformations to them in mathematical terms. This choice of intermediate representation—linear operators—inspires a whole set of transformations that we catalog here.

For purposes of explanation, we make a distinction between two classes of transformations based on their effect on leaf nodes in the operator tree. *Tree transformations* modify the internal structure of the tree but do not change the leaf nodes; in contrast, *realization transformations* alter the makeup of leaf nodes within the tree. We explain each in the following sections.

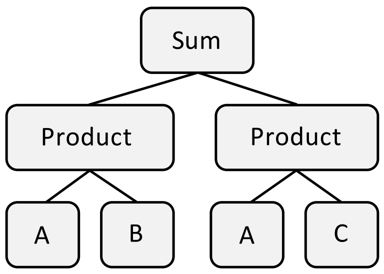
Tree Transformations

Like abstract syntax trees in general purpose programming languages, an Indigo AST is a data structure that can be modified to alter the program it represents. Modifications must preserve program semantics, but due to Indigo's rich intermediate representation, decisions about correctness can be reasoned about in mathematical terms.

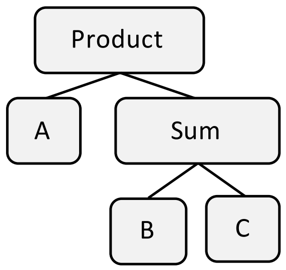
Indigo provides a number of linear operators classes that compose child operators into larger, more complex operators. These operators include arithmetic operators (sum, product, and Kronecker product) and structural operators (vertical/horizontal/diagonal stacks) and are described in detail in Section 3.2. These operators, when used in Indigo programs, appear as non-leaf nodes in a operator tree. When multiple intermediate nodes are nested together, there are often opportunities to re-arrange the operator tree according to mathematical properties. This benefit works in the other direction, too—we can use mathematical properties to generate a set of valid transformations. In exact arithmetic, these transformations are always correct. However, correctness isn't guaranteed n IEEE 754 floating-point arithmetic because floating point arithmetic is generally non-associative—operations can't be reordered. We instead recognize that linear transformations are well-behaved numerically, so the residual floating point error will be within a constant factor of machine epsilon.

Arithmetic Tree Transformations

Elementary mathematical properties can induce tree transformations in arithmetic expressions involving matrices. Consider the expression $A(B + C)$. We can apply the distributive property of multiplication over addition to yield the equivalent expression $AB + AC$.



(a) AST for $AB + AC$.



(b) AST for $A(B+C)$.

When evaluated, these different expression trees will possess different performance characteristics—the left-hand tree will perform one matrix-matrix multiplication, and the right-hand tree will perform two. For now, we avoid any assertions about which performs better and merely treat them as equivalent programs with different performance characteristics.

The distributive property is broadly applicable to our set of operators. Additional examples of tree equivalences enabled by the distributive property include:

- Products distribute over Sums: $(A + B)(C + D) = AB + AC + BC + BD$.
- Adjoints distribute over Sums: $(A + B)^H = A^H + B^H$.
- Adjoints distribute over Products: $(AB)^H = B^H A^H$.
- Products distribute over Kronecker Products: $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$.
- Kronecker Products distribute over Sums: $A \otimes (B + C) = (A \otimes B) + (A \otimes C)$.
- Kronecker Products distribute over Adjoints: $(A^H \otimes B^H) = (A \otimes B)^H$.

Operators also possess straightforward commutative, identity, and inverse properties:

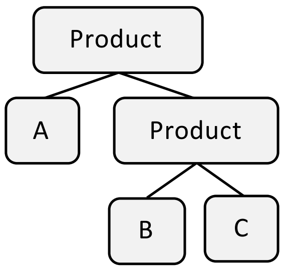
- Sums are commutative: $A + B = B + A$.
- Sums admit an identity operation: $A + 0 = A$.
- Products admit an identity operation: $A \cdot I = A$.
- Adjoints are involute: $(A^H)^H = A$.

Absent from this list is commutativity property of Product operators, which follows from the non-commutativity of matrix multiplication. Similarly, Kronecker products do not commute.

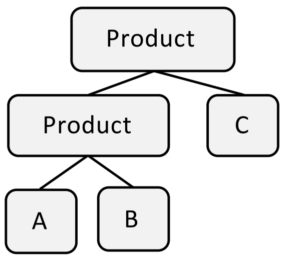
Lastly, Sum and Product operators are associative. This property allows for *tree rotations* which alter the internal structure of the tree but preserve the order of the leaves. For example, the equivalent expressions

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

result in the trees:



(a) AST for $A{\cdot}(B{\cdot}C)$.



(b) AST for $(A{\cdot}B){\cdot}C$.

Structural Tree Transformations

Our final class of tree transformations arises in the structural operators (vertical/horizontal/diagonal stacks). Here, it is sometimes possible to distribute an arithmetic operator over one or more structural operators. Common examples of structural tree transformations include:

- Dot-product-like Product:

$$\begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} C \\ D \end{bmatrix} == \begin{bmatrix} AC + CD \end{bmatrix}$$

- Matrix-vector-like Product:

$$\begin{bmatrix} A & \\ & B \end{bmatrix} \begin{bmatrix} C \\ D \end{bmatrix} = \begin{bmatrix} AC & BD \end{bmatrix}$$

- Adjoint of a block-diagonal matrix:

$$\begin{bmatrix} A & \\ & B \end{bmatrix}^H = \begin{bmatrix} A^H & \\ & B^H \end{bmatrix}$$

- Adjoint of a vertically-stacked matrix:

$$\begin{bmatrix} A \\ B \end{bmatrix}^H = \begin{bmatrix} A^H & B^H \end{bmatrix}$$

- Product of block-diagonal matrices:

$$\begin{bmatrix} A & \\ & B \end{bmatrix} \begin{bmatrix} C & \\ & D \end{bmatrix} = \begin{bmatrix} AC & \\ & BD \end{bmatrix}$$

Structural transformation possess the additional constraint that matrix dimensions must comport for a candidate transformation to be valid. This condition is easy to check programmatically and, in our experience, matrix dimensions typically match (as a reflection of the higher-level problem structure they represent).

Some structural transformations are possible but cannot be represented with Indigo's set of structural operators. In principle, we can perform a transformation on an outer-product-like Product:

$$\begin{bmatrix} A \\ B \end{bmatrix} \begin{bmatrix} C & D \end{bmatrix} = \begin{bmatrix} AC & AD \\ BC & BD \end{bmatrix}$$

The resulting matrix has a block populating every position, i.e. it's dense. Since Indigo lacks an operator to represent a dense matrix of blocks, the output cannot be represented and the transformation is not currently supported.

Realization Transformations

The second major class of transformations is *realization transformations*. A realization transformation is the conversion of a matrix-free operator to the equivalent, explicit matrix operator. All matrix-free operators admit this transformation in principle, but only some of the possible realization transformations are implemented. Since realization transformations create sparse matrices from matrix-free operators, they typically hinder performance; however, they enable the application of other, more profitable transformations and thus serve a useful role in the transformation catalog.

Realizing Leaf Operators

Indigo implements several matrix-free operators that are instantiated as leaves in an operator tree. The Identity operator is trivially realized as a DIA-format sparse matrix with ones on the diagonal. Similarly, the OneMatrix operator is a dense matrix of ones.

The FFT operator is more complicated. It is certainly possible, but highly impractical, to realize an FFT operator as its DFT matrix. To understand why, consider implementing a Fourier transform of a three-dimensional signal with $n = 256$ samples per dimension. The FFT implementation requires $O(n \log(n))$ floating point operations and $\Omega(n^3)$ bytes of memory traffic. In contrast, the DFT matrix-vector product will require $O(n^6)$ flops and move $O(n^6)$ bytes (since the DFT matrix will have side length n^3). The reason for this discrepancy, and the insight behind the Fast Fourier Transform algorithm, is that the Fourier Transform contains redundancies in computation that are leveraged by the FFT but not the DFT matrix-vector product. Because of this large gulf in anticipated performance, and the presence of two- and three-dimensional Fourier transforms in computational imaging, Indigo does not implement realization of FFT operators.

Realizing Composite Operators

It is also possible to realize composite operators, not just leaf operators. Composite operators are eligible to be realized if and only if all descendants (children, grandchild, etc.) are realizable. The result of realizing a composite operator is a leaf operator—an explicit Matrix.

In order to realize a composite operator, the transformation engine must first realize its child operators. Then, it constructs a new Matrix operator according to the semantics of the operator being realized.

- Sum operators perform matrix-matrix addition. The resulting matrix implements the same transformation as the original Sum operator, but requires half the floating point operators and half the data movement (when the matrices are dense) or up to half (when the matrices are sparse).

- Product operators perform matrix-matrix multiplication. The performance characteristics of the product matrix depend on the dimensions of the original matrices, and if the matrices are sparse, the structure of the nonzero elements.
- Kronecker Product operators, when realized, form a new matrix in accordance with the definition of the Kronecker product.

$$A \otimes B = \begin{bmatrix} a_{1,1}B & a_{0,1}B & \dots & a_{1,n}B \\ a_{2,1}B & a_{1,1}B & & \vdots \\ \vdots & & \ddots & \\ a_{m,1}B & \dots & & a_{m,n}B \end{bmatrix}$$

- Adjoint operators realize their underlying matrix and apply a conjugate transpose to it.
- Structural operators realize their block matrices and combine them into a single matrix with the desired block arrangement.

The process of realizing operators is the machinery by which Indigo implements operator fusion. Operator fusion—merging adjacent tasks into one—is most similar to loop fusion in general purpose compilers, where complicated static analysis techniques have developed been employed to discover and leverage fusion opportunities [35, 10]. In Indigo, however, fusion is merely a mathematical operation, usually matrix-matrix addition or multiplication, because the domain of programming concepts is limited to linear operators. The analysis and transformation burden is dramatically reduced, and consequently Indigo can apply complicated fusion transformations with minimal programmer effort.

5.2 Profitability of Transformations

The previous subsections enumerated the transformations opportunities that might arise in an Indigo program without regard for the profitability of each transformation. In this section, we discuss the general implications of each class of transformations.

The purpose of a transformation, or set of transformations, is to achieve some desired performance characteristics. Typically, performance programmers seek to minimize execution time, but other objectives are possible: minimizing memory footprint, maximizing accuracy or quality, minimizing energy, or maximizing throughput. Often, these objectives can be combined; for example, a common objective for GPU programming is minimizing execution time subject to a threshold memory footprint (because it's desirable to leave data in GPU memory). When applying transformations to a program, it is important to assess each transformation's impact on the objective at hand.

Assessing the profitability of a transformation is not straightforward. In practice, we find that good program variants are often the result of a series of transformations to the original program given by the application programmer. These transformations are specified manually by a performance engineer in a Transformation Recipe, described momentarily in Section 5.3. Intriguingly, the performance of each program variant does not necessarily improve as successive transformations are applied. In practice, performance of the early variants is often worse than the initial variant, but as later transformations are applied, performance improves dramatically. This non-monotonic performance curve is a reflection of the need for "holistic," whole-program optimization.

It can be globally profitable to apply initial sub-optimal transformations because transformations can enable other transformations. For example, consider the operator C defined as the product of general matrix A and the sum of a general matrix B and an identity matrix I:

$$C = A(B + I).$$

We would like to realize this expression as a single matrix C'. However, the presence of the matrix-free Identity operator inhibits the realization of both the Product and Sum operators. To circumvent this condition, we realize the Identity matrix. This realization step is locally sub-optimal: we're replacing an operator with $O(1)$ memory footprint and zero floating point operations with a Matrix operator that requires $O(n)$ space and $O(n)$ flops (for side length n). However, it enables the realization of the Sum and Product operations, thus the overall storage, memory bandwidth, and computational burden decreases. In this example, the realization transformation is *transformation enabling* because, despite being locally suboptimal, allows other, more profitable transformations to proceed.

Many transformations have negligible effects on performance but enable more profitable transformations. We give two common examples:

- Realization of structural operators typically has minimal performance benefit. However, the realization allows fusion with adjacent matrices. Consider the operator:

  $$\begin{bmatrix} A & \\ & B \end{bmatrix} \begin{bmatrix} C \\ D \end{bmatrix}$$

  Realization of both the block-diagonal and vertically-stacked matrices enables fusion of the Product operator.

- Tree rotations, which apply the associativity property to rearrange trees, have minimal performance benefit when applied to series of Sums or square Products. They are useful, however, for grouping operators that can be realized, and for isolating operators that can't be realized. For example, consider an operator C defined as the Product of an FFT operator, F, and square matrices A and B:

  $$C = (F \cdot A) \cdot B.$$

  Under this arrangement, F inhibits any realization transformations in C. However, we can perform a tree rotation to associate A with B, and subsequently to fuse them together:

  $$C' = F \cdot (A \cdot B).$$

These examples demonstrate that the profitability of transformations isn't local, but must be considered in a global sense.

Additional profitability questions arise when fusing Matrix operators. Based on the dimensions of the participating matrices, the fused matrix may be much larger (in a outer-product like fusion) or much smaller (in an inner-product-like fusion). The situation is more complex when one or both matrices are sparse. With sparse matrices, fusion could produce matrices with any sparsity fraction, including fully-sparse (empty) and fully-dense matrices. In practice, matrices tend to maintain sparsity because they reflect structure in the application domain.

5.3 Transformation Recipes

The previous two sections presented the set of transformations and their performance implications. What remains to be explained is the process by which a set of transformations is chosen and applied. To accomplish the latter, we define an entity called a *transformation recipe* that represents the transformations to be applied to an Indigo program. Transformation recipes are procedural in nature, and the Indigo compiler applies them in their given order.

Choosing a set of transformations the constitute a transformation recipe remains a hard problem. We identify three avenues for selecting profitable transformations: expert authors, automatic methods, and generic recipes.

Recipes Authored by Expert Programmers

Programmers can directly write a transformation recipe based on transformations that they deem profitable. This approach is best suited for performance engineers since it allows for precise control over the resulting operator tree. The concise, declarative nature of transformation recipes enables rapid iteration with minimal code changes.

Relying on experts to author transformation recipes does not scale because it would require an expect produce a custom recipe for each operator.

Recipes Authored by Machines

Automatic techniques can search for profitable transformation recipes. Since machines can search faster than human programmers, they can potentially discover better recipes.

We briefly explored this avenue by casting recipe generation as a search problem and using the state-of-the-art search methods in the OpenTuner framework [2] to empirically evaluate thousands of different tree configurations. This approach failed to discover any recipes that bested the one we devised manually, so we did not develop it further.

In our experience, automatic search methods fail to find good recipes because of the irregularity of the search space. From our experience with expert-authored recipes, we found good configurations after a long series of transformations with minimal or negative effect on performance. Automatic methods that follow gradients didn't push through the flat or uphill regions that reflect the performance after early transformations, and thus couldn't access the highly-profitable later transformations.

Generic Recipes

In implementing a number of transportation recipes for various imaging applications, we noticed similarities between many of the recipes. In particular, good transformation recipes seemed to:

- Group operators that can be realized, and isolate ones that can't.
- Avoid realizing Kronecker Product operators.
- Conjugate-transpose tall-skinny matrices and store them under Adjoint operators.
- Minimize the overall number of operators.

Based on these observations, we suspect that it may be possible to write generic transformation recipes that yield decent performance across a broad range of imaging operators, without requiring expert knowledge on the part of the application programmer.

Chapter 6

Case Studies

This chapter presents applications of the Indigo DSL to a variety of image reconstruction problems in scientific and medical imaging. For each application, we assess the degree to which Indigo achieves the design goals given in Section 1 by presenting:

- a brief primer on the application,
- a description of the application's forward operator,
- a representation of the operator in the Indigo DSL,
- and performance results showing either near-peak performance, or sufficient performance that further optimization isn't required by application deadlines.

In this section, we assess Indigo on both quantitative performance metrics and qualitative goals including productiveness, expressiveness, robustness, and performance transparency. Previous literature has tried to quantify these qualitative goals, but such arguments are unconvincing in our experience. Rather than pursing zealous quantification, we choose to describe in detail the impact of Indigo design choices as they manifest in each application. We intend that a broad range of case studies will convince experts of the efficacy of our approach better than shallow quantitative metrics can.

This section begins with three applications to MR image reconstruction, including Cartesian, non-Cartesian, and T2-shuffled acquisitions. Since Indigo was heavily motivated by MR image reconstruction, these case studies include the most complicated and most optimized operators. Next, we present an application of Indigo to a reconstruction method found in x-ray ptychography. The ptychography operator falls slightly outside the domain of our domain-specific language, so this case study provides insight into the generality of our approach and the semantic degradation that occurs when working just outside the boundaries of the target application domain. Then, we assess Indigo on an operator from Magnetic Particle Imaging that has never before been implemented on a high-performance platform, and led to the development of a new operator class in the language. Finally, we cover two

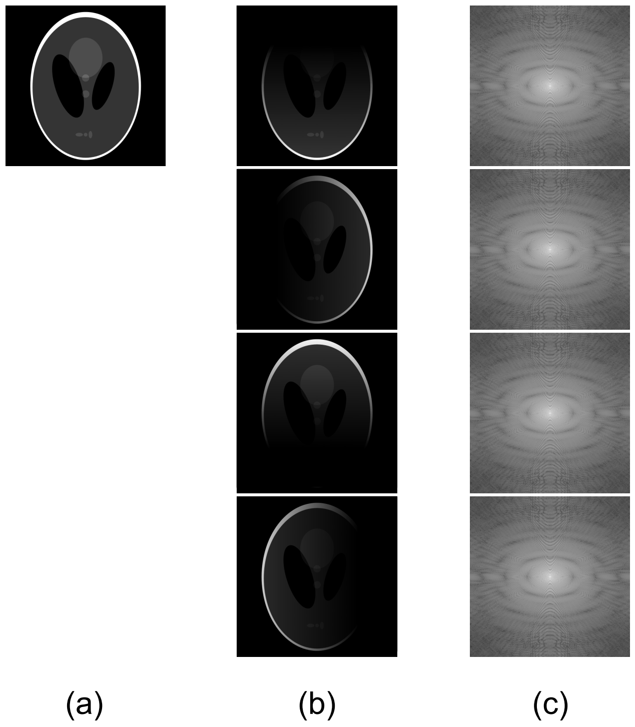


Figure 6.1: Illustration of the intermediate images in the SENSE operator for Cartesian acquisitions. The intrinsic image (a) is observed by multiple receive elements, each of the which observes a spatially-weighted version (b) of the intrinsic image. Then, the spatial encoding code is applied, effectively computing the Fourier Transform of each image, yield the images in (c). These continuous-valued Fourier images signals are then sampled at regular intervals to yield the data images acquired by the scanner.

cases from optical microscopy—phase space imaging and motion deblur microscopy—that dramatically benefit from high-performance implementations.

6.1 Cartesian MRI

MRI is a popular in-vivo imaging modality because of its excellent soft-tissue contrast, but its use is hindered by long scan times during which the patient must lie motionless.

Compressed sensing and parallel imaging techniques are being used to reduce scan times by integer factors, but reconstruction of the resulting data requires solving a linear inverse problem in a clinically-acceptable three minutes [41, 65]. A pure Python implementation can take as long as eight hours to produce a diagnostic-quality image, so we turn to Indigo to implement a fast reconstruction code.

During a scan, the scanner plays a "pulse sequence" of radio waves that excite protons or other species of interest with a particular volume. Additional magnetic fields induce spatially-

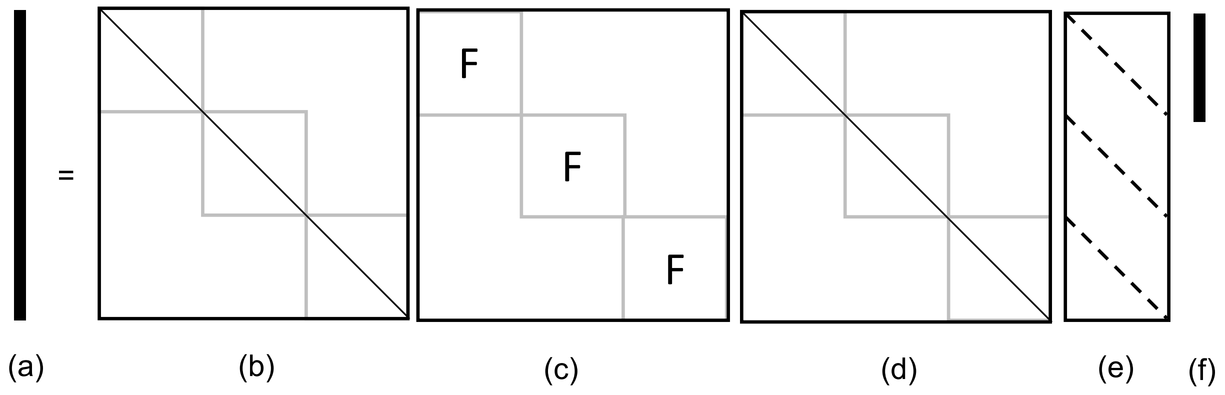


Figure 6.2: The Cartesian SENSE forward operation illustrated as a linear algebra operation. The intrinsic image, represented as the vector (f), is multiplied by a matrix, or series of matrices, to yield the acquired data, represented as the vector (a). The matrices correspond to major operations in the SENSE forward model—(e) applies the sensitivity maps, (d) performs an FFT-shift, (c) applies a batch discrete Fourier Transform, and (b) performs a second FFT-shift. Note that this factorization of the complete operator is one of many possible factorizations.

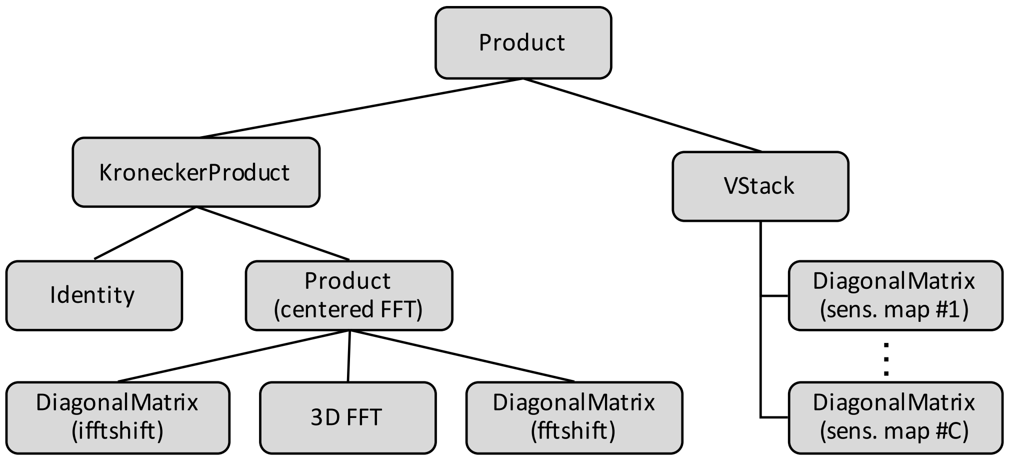

Product
KroneckerProduct
VStack
Identity
Product (centered FFT)
DiagonalMatrix (sens. map #1)
DiagonalMatrix (ifftshift)
3D FFT
DiagonalMatrix (fftshift)
DiagonalMatrix (sens. map #C)

Figure 6.3: The abstract syntax tree for the Cartesian SENSE operator employs a VStack operator to arrange the sensitivity maps vertically. The left half of the operator applies an centered FFT to each channel-image. Because the operator is identical across channels, we use a Kronecker Product operator with an Identity matrix argument to induce replication of the centered FFT operator. At runtime, the Indigo runtime specializes this construction and evaluates efficiently it as matrix-matrix multiplication.

varying resonant spinning of the protons, effectively encoding their position as frequency and aggregate density as amplitude. One or more inductive coils arranged around the volume of interest record the emitted signal in separate receive channels, and a Fourier transform of the channel data yields spatially-weighted variants of the intrinsic image. Compressed sensing pulse sequences speed up acquisitions by collecting fewer measurements than typically necessary. Linear reconstruction will result in aliasing in the spatially-weighted images, but the aliasing can later be resolved via knowledge of the spatial sensitivity of each receive coil and redundancies in the image statistics. In a mathematical optimization setting, we are looking for an image that most closely transforms into the aliased images acquired by the scanner.

We use the SENSE operator [53] to model how an image is transformed and acquired by the scanner. In SENSE, a candidate image ρ with shape (x, y, z) is first multiplied by c sensitivity maps $S_1, S_2, ...S_c$ of the same dimensions, yielding an array of channel-images with shape (c, x, y, z). Then, each channel-image undergoes a Fourier transform F to yield the acquired data k. Mathematically, the forward SENSE operation is

$$k_i = FS_i \cdot \rho, \quad \forall i \in c \tag{6.1}$$

and the adjoint operation is

$$\rho = \sum_{i=1}^{c} S_i^H \cdot F^H \cdot k_i. \tag{6.2}$$

Figure 6.1 illustrates the intermediate images in the Cartesian MRI forward operator.

Our application code generates a representation of the SENSE operator in Indigo. Given an NUFFT operator $\mathscr{F}$ and sensitivity maps stuffed into diagonal matrices S, the SENSE forward operator is expressed in matrix form as

$$(I_c \otimes \mathscr{F}) \cdot VStack(S_0, S_1, ..., S_c).$$

The Indigo operator tree is illustrated in Figure 6.2, and its corresponding abstract syntax tree is illustrated in Figure 6.3.

6.2 Non-Cartesian MRI

MRI scans can utilize more general scan trajectories than the Cartesian approach presented above. In this case, the input signal is sampled at non-regular positions in k-space and stored in a sparse data structure. The SENSE algorithm extends to the non-Cartesian setting with a few additional operators to handle the irregular sampling pattern.

The non-Cartesian SENSE operator was presented in Section 2.2. We contrast it briefly here the Cartesian variant just presented. The core of the non-Cartesian SENSE operator is a Cartesian SENSE operator with operators for sensitivity-weighting and centered FFTs; however, the major difference is the FFT is replaced with a non-uniform FFT (NUFFT)

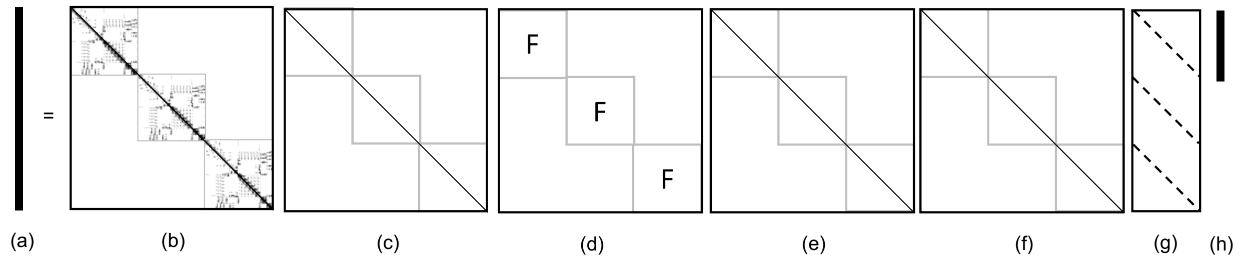


Figure 6.4: The non-Cartesian SENSE operator gives the relationship between the proposed image (h) and the acquired data (a). The image is first multiplied by sensitivity maps (g), then pre-apodized and shifted to the center (e), transformed via an FFT (d), un-shifted (c), and re-sampled at non-Cartesian points (b). Note the block-diagonal structure of most of the operators—this suggests channels can be processed in parallel. Furthermore, the matrices contain many repeated blocks down the diagonals, so this operator is a good candidate for Kronecker Products that can represent repetition efficiently.

to re-sample the signal from Cartesian coordinates to the coordinates sampled during the scan trajectory. In a matrix-driven implementation, such as in Indigo, the extra sampling operation can be represented as a *gridding* matrix and applied via a sparse matrix-matrix multiplication.

We further extend the SENSE operator to the multi-timepoint setting in which we seek to reconstruct a series of images taken over time. The multi-timepoint, non-Cartesian SENSE forward operator gives the following relationship between the intrinsic images ρ_t and the sparse image acquired by receive channel c at timepoint t:

$$k_{c,i} = P_t \mathcal{F}_t S_c \cdot \rho_t, \quad \forall C \in C, t \in T \tag{6.3}$$

As before, P_t and $\mathcal{F}_t$ represent the density compensation factors and the NUFFT, respectively. However, each timepoint now has a unique gridding operator stemming from the unique sampling trajectory realized during that timepoint's acquisition. One possible AST for the non-Cartesian, multi-timepoint SENSE operator is illustrated in Figure 6.5.

From a performance perspective, the multi-timepoint non-Cartesian SENSE operator is interesting because of opportunities for locality and parallelism in both the time and channel dimensions. For locality in particular, the same sensitivity map can be used across all timepoints for a given channel index. Likewise, the same gridding operator can be used for all channels within a timepoint. For parallelism, each channel-image can be computing independently of other timepoints and channels. This abundant parallelism suggests that it may be feasible to maximize locality in a forward operator evaluation. The opportunities for parallelism and locality can be visualized in a structural representation of the non-Cartesian SENSE operator. Figure 6.4 illustrates the operator for a three-channel reconstruction task.

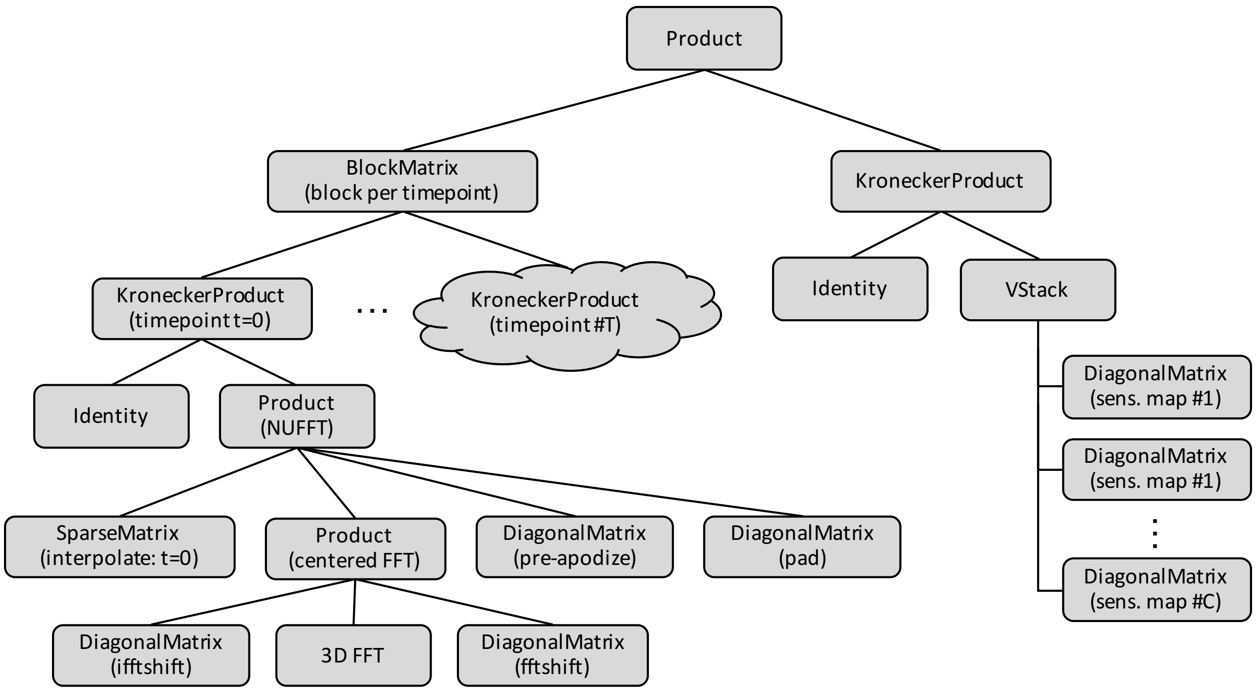
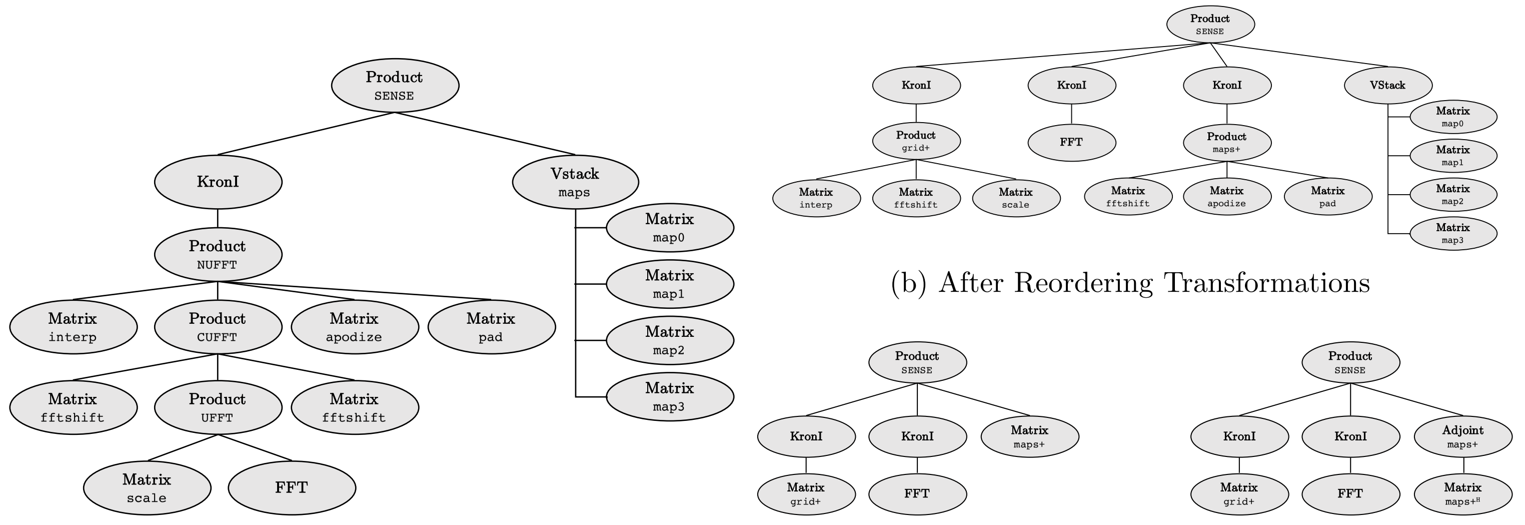

Product
BlockMatrix (block per timepoint)
KroneckerProduct
KroneckerProduct (timepoint t=0)
...
KroneckerProduct (timepoint #T)
Identity
VStack
Identity
Product (NUFFT)
DiagonalMatrix (sens. map #1)
DiagonalMatrix (sens. map #1)
DiagonalMatrix (sens. map #C)
SparseMatrix (interpolate: t=0)
Product (centered FFT)
DiagonalMatrix (pre-apodize)
DiagonalMatrix (pad)
DiagonalMatrix (ifftshift)
3D FFT
DiagonalMatrix (fftshift)

Figure 6.5: AST for the Multi-Timepoint SENSE Non-Cartesian Operator.

Transformation of the SENSE AST

The user-defined operator tree (Figure 6.6a) is functionally correct but slow to execute for various reasons. First, evaluating each of the many `MatrixOp` nodes requires reading and writing its input and output vectors, which is expensive in aggregate. We attempt to merge some of the these nodes to avoid intermediate memory traffic, but the matrix-free `FFT` node inhibits this transformation. Thus, we develop a transformation recipe that separates the `FFT` from concrete matrices by flattening nested `Products` and distributing the `KronI` operator over its child `Products`. The resulting tree is depicted in Figure 6.6b. Then, we realize `VStack` and `Product` nodes into new `MatrixOps`. We are left with two sparse matrices: a *grid+* matrix that performs gridding, shifting, and scaling; and a *maps+* matrix that simultaneously applies sensitivity maps, apodization, padding, and FFT shifting to each coil-image. The resulting tree in shown in Figure 6.6c.

A performance analysis of the tree in Figure 6.6c reveals that the sparse *maps+* matrix under-performs because the CSR storage format inhibits re-use of the input vector. Because the matrix has 0 or 1 nonzeros per row, and exactly $c = 8$ nonzeros per column, a row-wise parallelization (as is typical for CSR formats) will have difficulty exploiting temporal reuse of the input vectors. We can remedy this by explicitly conjugate-transposing the matrix and inserting an adjoint node, yielding the final tree depicted in Figure 6.6d. Storing the adjoint matrix in CSR format is equivalent to storing the original matrix in compressed sparse column (CSC) storage format, but only requires that backends implement CSR multiplication



(a) User-Provided Operator Tree

(b) After Reordering Transformations

(c) After Operator Realization

(d) After `maps+` Transpose

Figure 6.6: We apply a series of transformations to the SENSE operator tree to produce a high-performance variant. The user-provided tree (a) is first reordered to lift matrix-free FFT nodes by distributing the Kroneck Product operator over its child Products, yielding the tree in (b). Then, matrices are realized in (c) and the 'maps+' matrix is transposed in (d). Evaluation of the final tree requires two sparse matrix-matrix multiplications and one batched FFT.

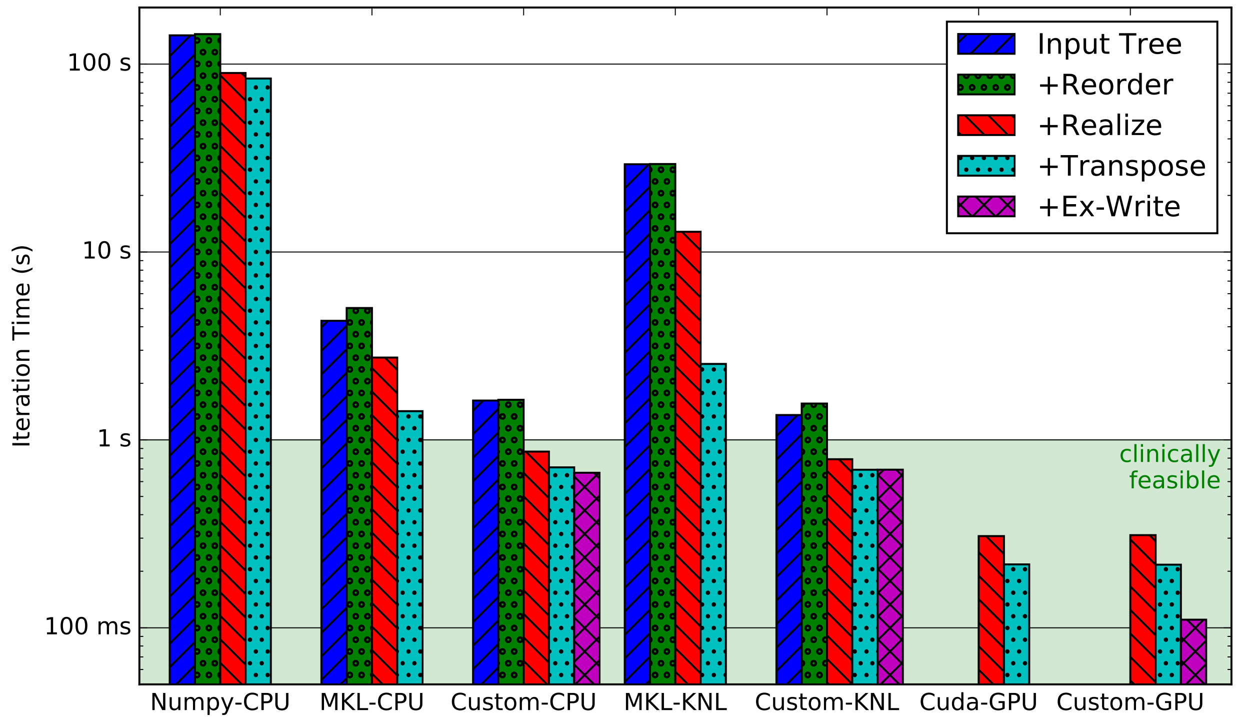


Figure 6.7: This plot illustrates how the performance of the non-Cartesian SENSE operator changes as our prescribed transformations are applied: the input tree (Figure 6.6a), with re-ordering transformations (Figure 6.6b), with most operators realized as explicit matrices (Figure 6.6c), with the `maps` matrix transposed (Figure 6.6d), and with exploitation of the exclusive-write property of the `maps` matrix. Results indicate that the best ASTs perform up to ten times better than straightforward trees. On the GPU, the transformations are essential to reducing the memory footprint enough to be device-memory resident. Finally, only with the transformations were we able to achieve our goal of clinical feasibility—1 iteration per second—on all three platforms.

routines.

The remaining performance discrepancy arises on GPUs, where the atomic accumulate operations found in adjoint CSR multiplication kernels are particularly expensive. Here, we leverage the row-wise exclusive write property of the `maps+` matrix to dispatch to sparse matrix-matrix multiplication routines that perform non-atomic accumulations, rather than atomic ones. This results in a 10× speedup on the adjoint evaluation of the `maps+` matrix, and a 2× speedup overall.

		FFTs		SpMMs		Overall		
Platform		%Peak	%Time	%Peak	%Time	GFlops/s	%Peak	Time
CPU	Numpy	0%	86%	0	14%	1	0%	87,040 ms
	MKL	41%	31%	1	69%	51	19%	1,321 ms
	Custom	40%	67%	4	33%	107	39%	743 ms
KNL	MKL	11%	13%	0	87%	27	2%	1,573 ms
	Custom	11%	59%	4	41%	124	10%	536 ms
GPU	CUDA	95%	34%	8	66%	327	44%	205 ms
	Custom	95%	72%	37	28%	680	91%	114 ms

Table 6.1: Performance breakdown of one iteration of the non-Cartesian SENSE normal operator ($A^H A$). The fraction of peak performance is measured with respect to the Roofline peak for each operator. The best platform achieves 91% of the Roofline peak. These data suggest the use of explicit sparse matrices isn't a significance hindrance since the majority of the evaluation time (59%-72%) is spent in vendor-tuned FFT routines.

Detailed Performance Results for the Non-Cartesian SENSE Operator

The Indigo SENSE operator achieves 91% of the Roofline peak on the GPU platform, and the reconstruction task finishes within one minute. Table 6.1 gives the performance breakdown of the fastest AST across the suite of platforms. The CPU and KNL platforms achieve a smaller fraction of peak—33% and 10%, respectively. We still consider this a worthy result because two-thirds of the evaluation time is spent in the vendor-tuned FFT library (MKL) using FFT dimensions suggested by the vendor's tuning script. Assuming the MKL FFT is well-implemented, and recognizing that optimizing it is beyond our control, we conclude that our CPU and KNL backends are achieving a reasonable fraction of peak performance. On all platforms, our best-performing backend is within the range of clinical feasibility.

Comparison to Matrix-Free SENSE

An interesting comparison can be done between the Indigo implementation of the non-Cartesian SENSE operator and a conceptual, well-optimized, matrix-free implementation. Indigo's matrix-stuffing approach is non-optimal from a performance perspective, and we sketch here an argument that attempts to quantify the degree of performance degradation with respect to a matrix-free implementation.

We can characterize the performance of a fully-matrix-free SENSE implementation if we make two assumptions: 1) the intermediate vectors are memory-resident before and after the FFTs. To do otherwise would be to ignore re-use in the `maps+` and `grid+` matrix-free operators, and 2) the sensitivity maps are incompressible, i.e. they do not permit more succinct representations than dense arrays. Given these assumptions, we can quantify how much faster each operation could be:

- A matrix-free `maps+` evaluation still must read the sensitivity map values, but it can infer index data rather than read it from the sparse matrix data structure. In a reconstruction with $c = 8$ channels, each 8-byte `maps+` nonzero reads one integer for the column index, 1/8 integer for the row index, and 1/8 input elements, and writes 1 output element; thus $4.125/(25 + 4.125) = 14\%$ of the memory traffic can be avoided. Since `maps+` evaluations make up 17% of the iteration time on our best platform, this would yield a 2.4% overall speedup.

- The FFT operation is matrix-free in both cases and its performance is identical.

- A matrix-free `grid+` operator could generate both indices and non-zero coefficients on the fly, so the only cost would be reading and writing the vectors. Since `grid+` evaluations comprise only 12% of the overall iteration time, we conservatively assert that a matrix-free gridding operator will yield no more than a 12% speedup.

Taken together, these results indicate that the Indigo implementation strategy can achieve at least 85% of the performance of an optimal matrix-free implementation. This result can also be interpreted as the marginal cost of the CSR storage format. Increasingly specialized storage formats and their associated evaluation routines can further reduce memory traffic.

6.3 T2-Shuffled MRI

Recent work by Tamir and et al [60] proposes a new method for acquiring multiple images at different contrasts. Their technique modifies the SENSE forward operator to include a special matrix that “shuffles” samples between those acquired across time in the scan and those acquired at different signal decay values (which provide a source of image contrast). In this brief case study, we identify how Indigo could implement the T2-shuffling operator.

The T2-shuffling operator differs from the standard SENSE operator in the inclusion of a shuffling matrix to transform between the proposed contrast images and and the acquired time images. In the original work, this operation manifests as a right-hand-side multiplication of the contrast images by the shuffling matrix, as illustrated in Figure 6.8.

At first glance, representing the right-hand matrix multiplication falls outside the domain of Indigo’s linear operator abstraction. However, we can apply a useful property of the Kronecker Product to express right-hand multiplication of the contrast images X and the system matrix A as left-hand multiplication:

$$XA = (A^H \otimes I) \cdot vec(X).$$

This yields the operator structure illustrated in Figure 6.9.

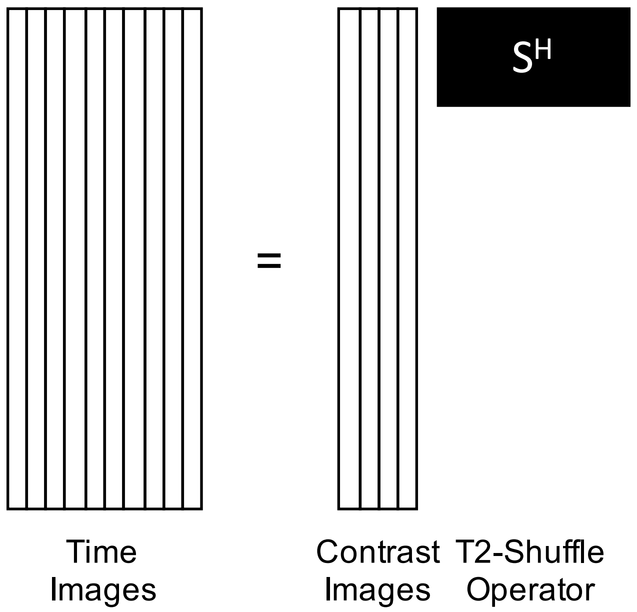


Figure 6.8: The T2-Shuffling operator models the relationship between a large set of images acquired at different timepoints, and a small set of computed images acquired at different T2 decay values.

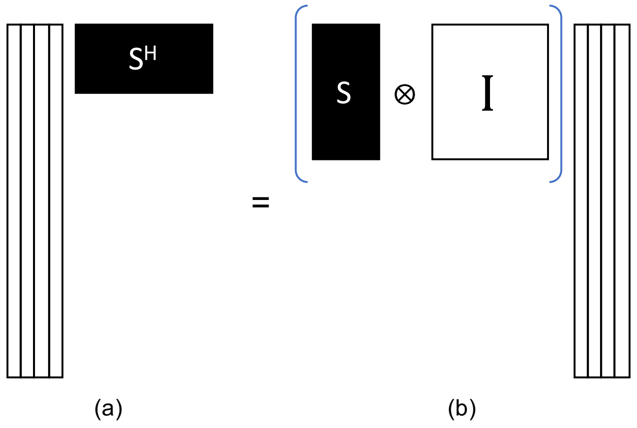


Figure 6.9: Formulation of the T2-Shuffling Step as a Left-Hand Multiplication

We did not implement the T2-Shuffling variant. Because the shuffling operator is typically small, it's performance is likely negligible and the T2-Shuffling SENSE operator's performance should match that of the standard SENSE operator.

6.4 Ptychography

Ptychography is an imaging modality which uses patterns from diffracted light to reconstruct an image of an object [43]. In X-ray ptychography, X-rays are shined through an object, ψ, with some illumination pattern ω. The X-rays shine through the object to land on a detector, such as a CCD sensor. A number of diffraction samples $a_{(i)}$ are gathered by shining the X-ray beam at the object from different angles. The diffraction of the X-rays can be described by the equation

$$\mathbf{a} = |\mathbf{FQ}\psi^\vee|$$

where $\psi^\vee$ is a vector holding a linearized version of the object being imaged, $\mathbf{Q}$ is an operator which extracts frames from the image and scales them by the illumination ω, $\mathbf{F}$ is a Kronecker product $(I_K \otimes \mathscr{F})$ of K two-dimensional Fourier transforms, and $\mathbf{a}$ is a vector holding a linearized version of the diffraction samples captured during the experiment. We can refer to frames extracted from the image $\psi^\vee$ as $\mathbf{z} = \mathbf{Q}\psi^\vee$.

The full update step is given by

$$\mathbf{z}_{i+1} = \mathbf{Q}(\mathbf{Q}^H\mathbf{Q})^{-1}\mathbf{Q}^H\mathbf{F}^H\text{diag}(\mathbf{a})\frac{\mathbf{Fz}_i}{|\mathbf{Fz}_i|}.$$

Note that this includes the nonlinear term $|\mathbf{Fz}_i|$. Since nonlinearity is beyond the domain of Indigo, we instead represent the update operator as two separate linear operators bridged by a nonlinear operation. The linear operators can be optimized separately, and we are free to pick arbitrary nonlinear code to execute between them.

In order to achieve high performance, we construct a transformation recipe that shifts the tree to fold $\mathbf{F}^H$'s FFT scaling matrix into diag($\mathbf{a}$) and combines $(\mathbf{Q}^H\mathbf{Q})^{-}1$ with $\mathbf{Q}^H$. Table 6.2 shows the performance of this solution. Since matrix structure does not admit the exclusive write optimization, we do not list numbers for the custom backends. We achieve the best result, 76% of Roofline peak, on the GPU, followed by 56% of Roofline peak on the CPU. Our KNL version runs faster than the CPU version, but only achieves 9% of Roofline peak. The GPU version is 274 times faster than when executed in NumPy.

6.5 Magnetic Particle Imaging

Magnetic particle imaging (MPI) is a novel in-vivo imaging technique that seeks to acquire an image representing the distribution of a magnetic tracer within a volume of interest [26].

In a simplified sense, pixel data are acquired as a series of overlapping panels that represent subsets of the intrinsic image, but panels have had their average values removed by

Platform		FFTs %Peak	SpMMs %Peak	Overall %FFTs	Overall %Peak	Overall Time
CPU	Numpy	1%	2%	39%	1%	1,549 ms
	MKL	53%	59%	19%	56%	22 ms
KNL	MKL	13%	5%	10%	9%	35 ms
GPU	CUDA	92%	61%	34%	76%	6 ms

Table 6.2: Performance breakdown of one iteration of the Ptychography reconstruction operator (Section 6.4). Fraction of peak data are with respect to the Roofline peak for each respective operation.

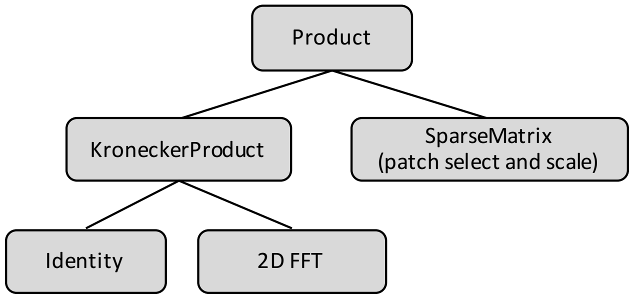


Figure 6.10: Abstract Syntax Tree for the ptychography forward operator.

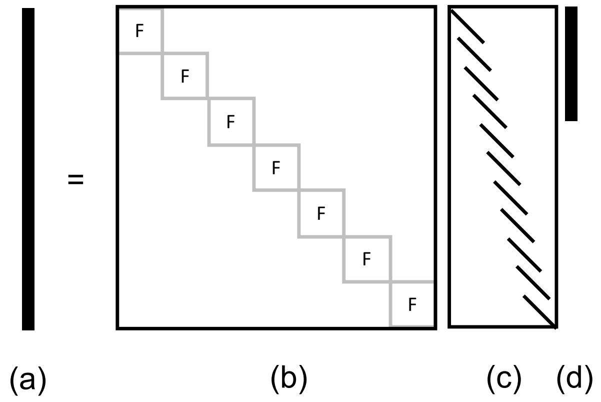


Figure 6.11: Illustration of the ptychography forward operator.

physical effects of the MPI scanner. The reconstruction objective is to modulate the average value of each panel so their overlapping sections are consistent. Konkle et al formulate this as an optimization problem with a non-negativity constraint [39]. They propose a forward operator $A = DS$, where S selects overlapping panels of the image and D computes and

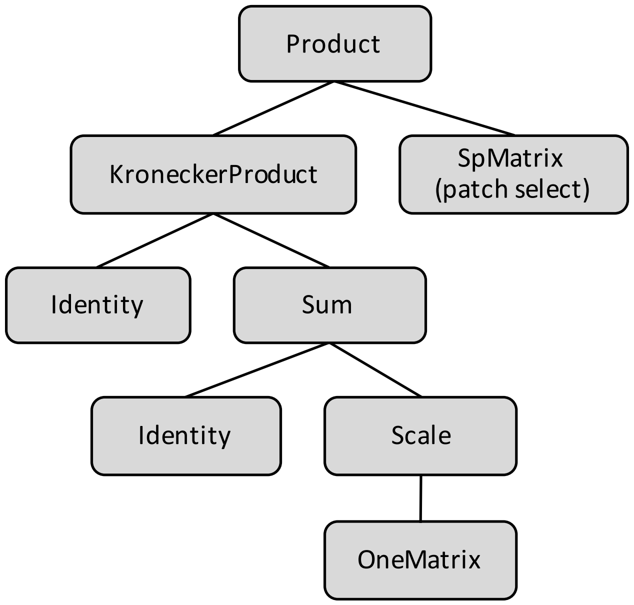


Figure 6.12: Abstract Syntax Tree for the Convex Magnetic Particle Imaging Operator.

subtracts the average value of each panel. Visually, the operators are

$$S = \begin{bmatrix} I_s & & & & & \cdots \\ & I_r & & & & \\ & & I_s & & & \\ & & I_s & & & \\ & & & I_r & & \\ & & & & I_s & \\ \vdots & & & & I_s & \ddots \end{bmatrix}, \quad \begin{array}{l} D = \begin{bmatrix} R & & \\ & \ddots & \\ & & R \end{bmatrix}, \\ \\ R = I_p - \frac{1}{p}\mathbb{1}. \end{array}$$

In this notation, $\mathbb{1}$ denotes a matrix of ones. Panels contain p pixels, overlap by s, and $r = p - 2s$.

In Indigo, we implement the S operator via matrix-stuffing, and the R operator via compositions of matrix-free operators. The MPI operator is different from our other applications because it doesn't perform any FFTs. Instead, the forward operator reduces to an Vector-vector multiplication $(\alpha X + \beta Y)$ (AXBY), a Sparse Matrix times Dense Matrix Multiplication (SpMM) (with the S matrix), and a multiplication by a matrix of ones (OneMM). We evaluate the operator on a 1024-by-768 pixel image divided into 12 panels. Performance results are given in Table 6.3. The GPU achieves 43% of Roofline peak, indicating that a 2.3× speedup is possible, but at 1 millisecond per evaluation further optimization isn't motivated by application demands.

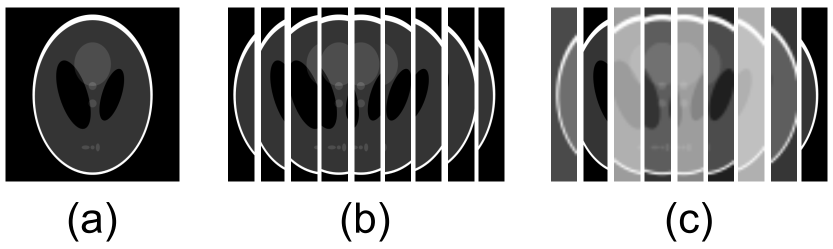


Figure 6.13: Intermediate images in the Magnetic Particle Imaging (MPI) forward operator. The operator has two major steps: first, the proposed image (a) is split into a number of overlapping panels (b), then each panel has its average value subtracted out yielding the panels produced by the MPI scanner (c).

		AXBPYs		SpMMs		OneMMs		Overall	
Platform		%Peak	%Time	%Peak	%Time	%Peak	%Time	%Peak	Time
CPU	Numpy	3%	16%	4%	31%	1%	53%	2%	91.43 ms
	Custom	34%	21%	30%	55%	17%	23%	26%	6.71 ms
KNL	Custom	4%	32%	6%	48%	5%	20%	5%	7.10 ms
GPU	Custom	34%	36%	69%	38%	23%	27%	43%	1.05 ms

Table 6.3: Performance breakdown of one iteration of the Magnetic Particle Imaging reconstruction operator (Section 6.5). Fraction of peak data are with respect to the Roofline peak for each respective operation. We omit results from the MKL and CUDA backends which don't implement the OneMM routine (a multiplication by a matrix of ones). The results indicate that Indigo achieves a modest fraction of peak (e.g. 43% on the GPU). It is likely possible to attain better performance (up to 2.3× on the GPU), but the current performance is sufficient to meet application needs.

6.6 Fluorescent Microscopy

Phase-space fluorescent microscopy enables 3D reconstruction of fluorescence by leveraging the position and angular information of light. Biologists often want to study the activity of a 3D creature or cells *in vivo*, hence the need to capture a 3D video. Both the fast acquisition of the phase-space information and fast 3D reconstruction from the data are important in order to visualize the sample. In recent work [40], multiplexed phase-space imaging aims to tackle this challenge. The slow mechanical scanning or angle scanning part is replaced by applying multiplex codes in the angular space (pupil). An image is captured for each of the codes while the sample is uniformly illuminated by a steady laser source, and the 3D sample is reconstructed from those images. The structure of data movement in the phase-space forward model is illustrated in Figure 6.14.

In the phase-space forward operator, a 3D volume is split into multiple 2D depth planes.

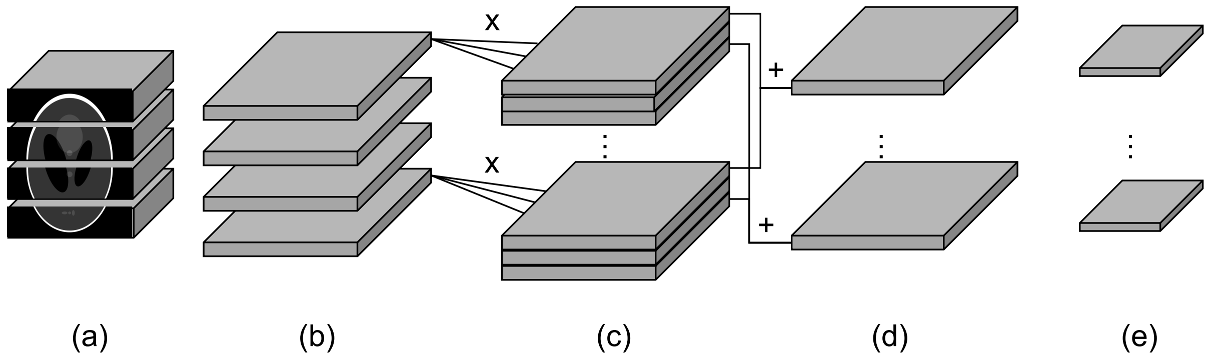


Figure 6.14: In phase-space imaging, we seek to reconstruct a three-dimensional volume comprising a set of two-dimensional slabs (e). In the phase-space forward model, each of these slabs is individually padded and Fourier-transformed (d), multiplied by an illumination map corresponding to the slab's depth (c), summed with contributions from other slabs in the same illumination instance (b), and inverse transformed and cropped to produce a series of two-dimensional images that are acquired by a microscope (a).

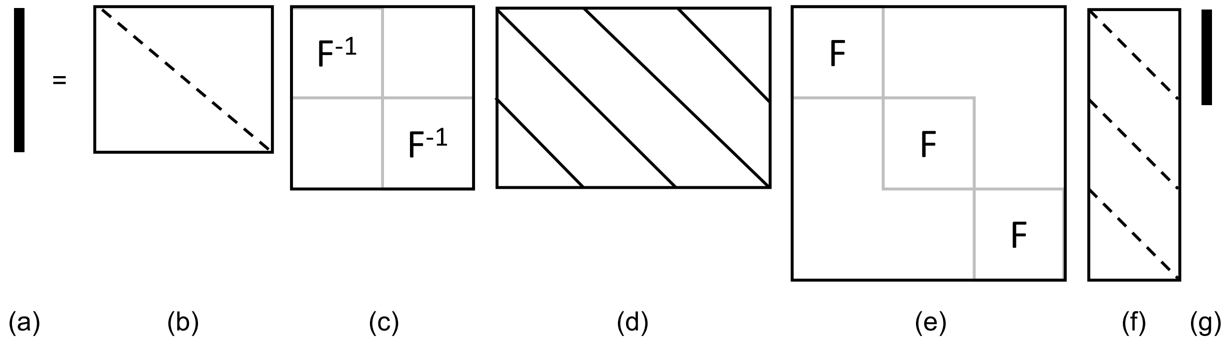


Figure 6.15: The phase-space linear operator translates a three-dimensional image (g) into a set of two-dimensional images acquired by the microscope (a). This is implemented by copying and padding the original images (f), Fourier transforming them (e), multiplying them by illumination patterns and summing images with the same illumination pattern (d), inverse transforming (c), and cropping (b).

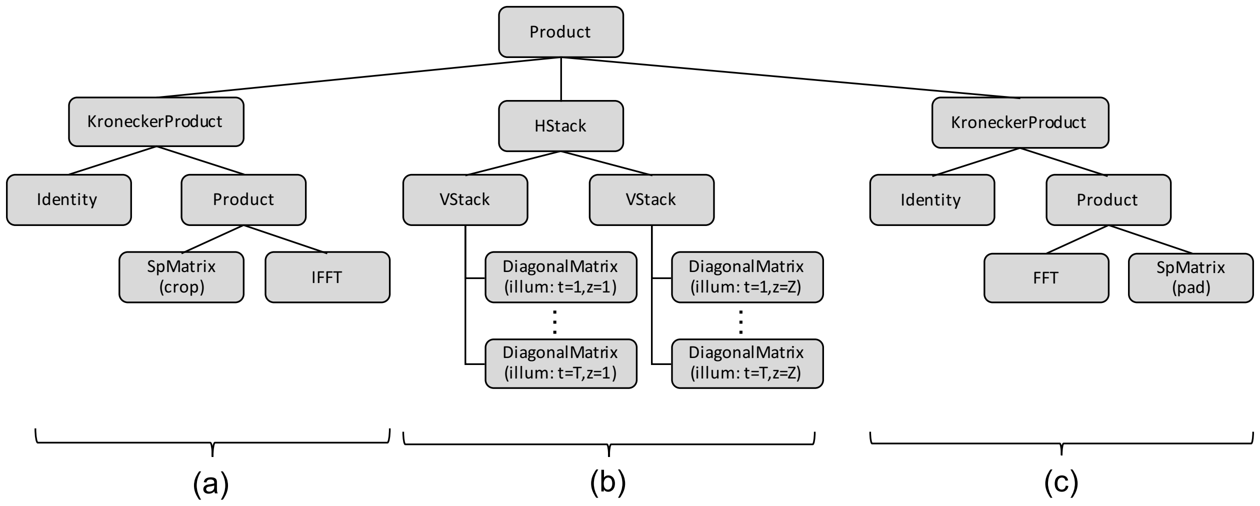


Figure 6.16: The phase-space operator AST is composed of three major sections. The input images are first padded and Fourier transformed (c), then the set of images undergoes a mixing scheme that multiplies each Fourier image by an illumination map and sums the images from the same illumination (b), and finally the image is inverse Fourier transformed and cropped to yield the images acquired by the microscope.

Then, each plane is padded around the edges, Fourier-transformed, and multiplied by a kernel related to a multiplex code and the corresponding depth (kernels are precomputed). The depth planes corresponding to a single code are summed, inverse Fourier-transformed, and cropped, yielding the images received by the microscope. This process is illustrated in Figure 6.14, and can be implemented by the linear operator with structure illustrated in Figure 6.15 and AST given in 6.16.

To devise a high-performance operator, our transformation recipe groups `SpMatrices` in the operator tree, aggressively realizes `CompositeOps`, and stores one of the two sparse matrices in a diagonal storage format and the other in CSR format. Our recipe reduces the phase-space operator to two batch FFTs interleaved with two SpMMs. We evaluate the high-performance operator on a reconstruction problem with five 1024^2 planes and 29 detection codes. Table 6.4 gives the operator's performance on each of our test platforms. On the GPU, Indigo achieves 47% of the Roofline peak and 186× the performance of the Numpy backend. Further optimization isn't motivated by the application, but we expect more tailored matrix storage formats would yield speedups in the future.

6.7 Motion-Deblurred Microscopy

A common task in microscopy is the acquisition of a high-resolution image of a large two-dimensional surface. However, optical constraints limit the microscope's field of view to a narrow region, so large acquisitions are typically performed as a series of small acquisitions.

		FFTs	SpMMs	Overall		
Platform		%Peak	%Peak	%FFTs	%Peak	Time
CPU	Numpy	1%	2%	66%	1%	17.9 s
	Custom	65%	36%	25%	43%	482 ms
KNL	Custom	8%	7%	35%	7%	597 ms
GPU	Custom	56%	41%	39%	47%	96 ms

Table 6.4: Performance breakdown of one iteration of the phase-space microscopy reconstruction operator. Fraction of peak data are computed with respect to the Roofline peak for each operation. The Indigo operator is within a factor of two of peak performance on the CPU and GPU platforms using our custom backends, and the best GPU implementation is 186× faster than the same operator running on a CPU via Numpy.

Here, the imaging device is typically rastered over the region of interest, pausing to acquire images in the so-called "stop and stare" approach. When many images must be acquired, this approach can be too time-consuming—some applications like the imaging of rapidly-decaying biological samples require must faster techniques.

An alternative to the "stop and stare" approach is constantly sweeping the imaging device across the region of interest. The latter approach is troubling because it incurs motion blur in the acquired images. However, by varying the illumination and posing the reconstruction task as an optimization problem, it is possible to recover the desired images to high fidelity. This technique is termed motion-deblurred microscopy [42].

We can model the motion deblur problem as a linear transformation. The motion blur operator takes as input a large image. It then blurs each image by transforming it to Fourier space, multiplying it by the illumination pattern, and transforming it back to image space. Finally, the images are cropped to represent the field of view for each small image acquired by the microscope. Figure 6.18 illustrates the intermediate images in an example forward operator.

The linear operator representing this transformation is illustrated in Figure 6.17. It utilizes a now familiar pattern: initial Fourier transforms, then Fourier-space multiplication, inverse Fourier transforms, and finally cropping. These operations can be represented in AST form in Figure 6.19. We apply realization transformations to the `VStack`, but otherwise run the operator as is. We did not perform a performance evaluation of the motion de-blur operator, but in our experience it was so fast that we didn't explore additional optimization efforts.

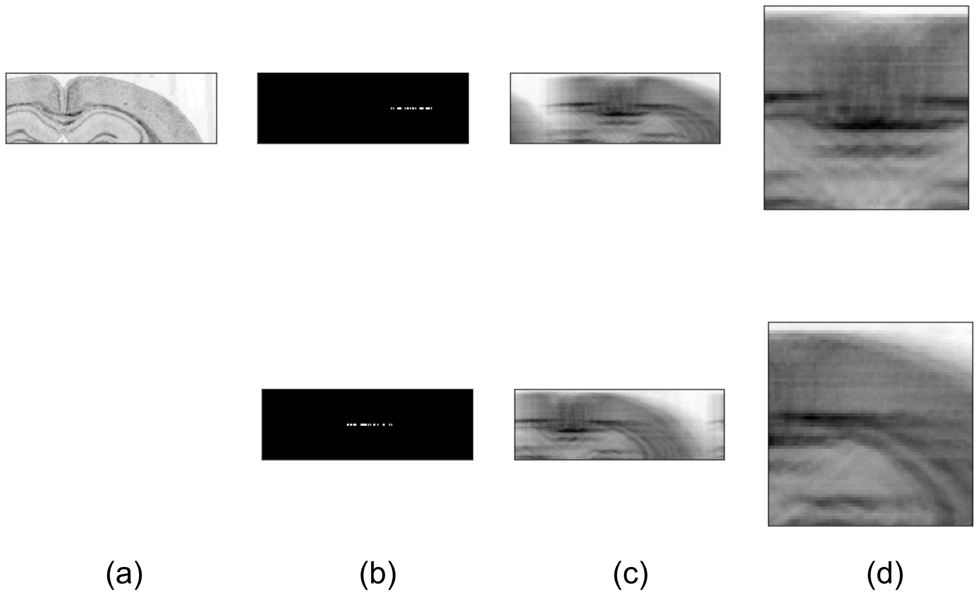


Figure 6.17: In the motion-deblur forward operator, the intrinsic image (a) is convolved with a kernel representing the illumination pattern (b), shown here in frequency space. The resulting image (c) is a translated, blurred version of the original. It is then cropped to the field of view of the microscope; the result is a set of images (d) that are acquired by the microscope. The images possess some overlapping pixels, hence reconstruction can be posed as a global optimization problem.

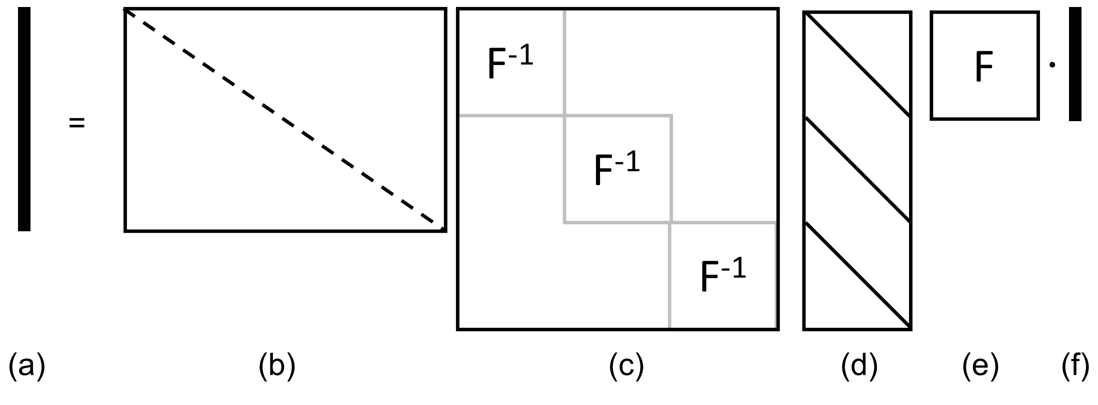


Figure 6.18: The linear operator representation of the motion de-blur application. Reading right-to-left: the proposed solution image (f) is convolved with multiple illumination patterns via multiplication in frequency space. The illumination patterns are vectorized and stacked in a multi-diagonal matrix (d) sandwiched between forward and inverse Fourier transforms ((e) and (c)). The resulting images are then cropped by (b) to produce the set of images acquired by the scanner (a).

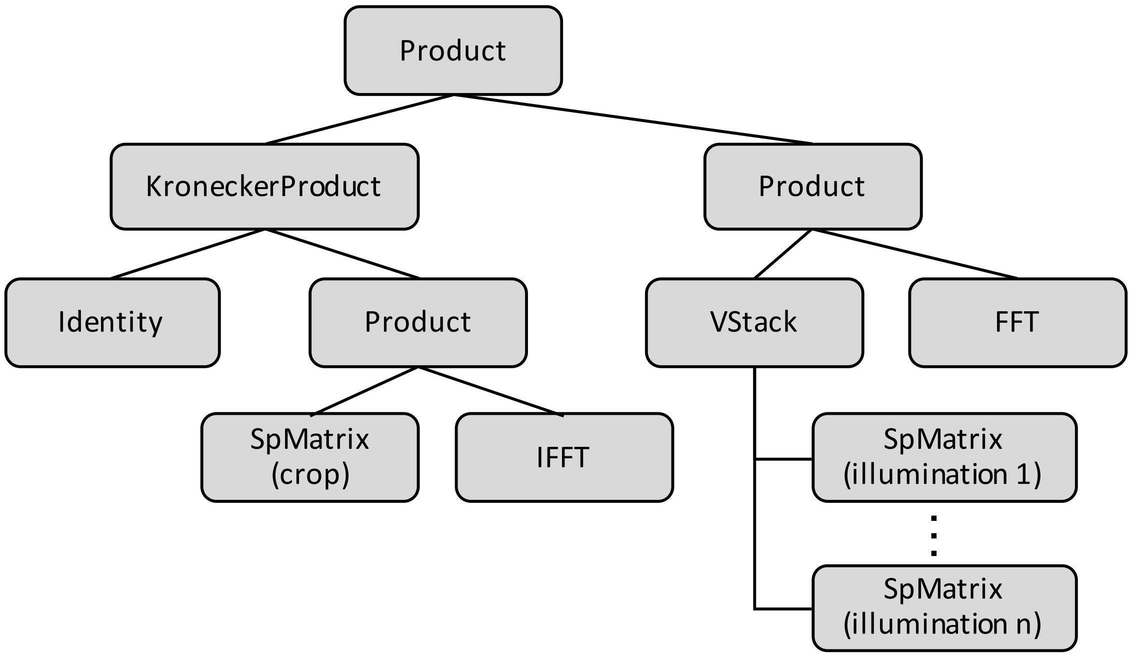


Figure 6.19: The motion de-blur operator implements a convolution via Fourier-space multiplication, and a crop operation on the resulting images. In our transformation recipe, we realize the `VStack` operator but otherwise run the operator as is. The operator was sufficiently fast that we didn't explore additional optimization opportunities.

Chapter 7

Extensions for Distributed-Memory

A handful of image reconstruction problems require more resources than shared-memory systems provide. There are two primary cases where this condition may arise. First, problem instances can require more high-bandwidth memory than a system provides—modern systems with typically provide up to 16 GB. Reconstruction problems that demand more space cannot run on entirely on modern GPUs, and on the KNL many-core system they spill to the larger, slower main memory and suffer a performance hit. Second, shared-memory systems may also fail to provide enough computational throughput to perform a reconstruction within a prescribed deadline. A well-optimized code is still limited by the throughput of the underlying processor.

If an application is using a reasonable algorithm and is well-optimized, but still not meeting its performance deadlines, a few options exist for continued performance improvements. The simplest technique is to wait for the next generation of processors, since Moore's Law is still delivering performance gains via parallelism, albeit with diminishing returns going forward. Another, more immediate option is to port the code to other, faster shared-memory processors, but often at substantial development cost and only for modest speed improvements. The development effort may be better spent on a third solution—porting the code to a distributed-memory machine. Distributed-memory machines are collections of processors connected by a network, each with their own private memory. Both the aggregate amount of memory and the computational throughput scales with the size of the machine, so they can solve problems that are too big for shared-memory platforms, or execute too slowly.

Although shared-memory systems suffice for most modern image reconstruction problems, we came across one problem that necessitated the use of a distributed-memory machine—regularized reconstruction of multi-timepoint MRI acquisitions. This chapter describes the problem in detail and presents a solution for scaling Indigo to distributed-memory machines. We give performance results from experiments on a modern supercomputer that indicate perfect performance scaling as both problem sizes and machine sizes grow.

7.1 Motivation

As described in Chapter 2, it is desirable for clinical MRI reconstruction to complete within a minute so that medical staff can verify the quality of the images before removing the patient from the scanner. Depending on the type of scan performed, this can pose a substantial computational challenge. The challenge is best exemplified by the University of Wisconsin ultra-short TE dataset (UWUTE), comprising 49M single-precision, non-Cartesian k-space samples for each of 8 acquisition channels, or 392M samples in total. The samples were taken continuously over time, so researchers binned them into 750 timepoints with 65.4K samples per channel. The 750 reconstructed volumetric images measure 208 by 308 by 480 pixels and total require 23GB to store. Clearly, this is too large to be resident in modern high-bandwidth memories.

The UWUTE dataset plausibly fits in standard memory, which has capacities into the terabytes, but such big, shared-memory machines do not provide enough processing power to solve the problem in a clinically-acceptable timeframe. A back-of-the-envelope calculation shows that distributed-memory is necessary for meeting the deadline. The standard SENSE algorithm requires an FFT and an IFFT per channel per timepoint per iteration. Using 8 channels, 750 timepoints, and 100 iterations, the reconstruction requires 600K FFTs and 600K IFFTs. From a performance perspective, FFTs and IFFTs are identical, so we seek the time required to computer 1.2M FFTs. Using the arithmetic intensity of 2.59 flops/byte for an FFT (derived in Section 2.4) and a STREAM memory bandwidth of 102 GB/s, we estimate the peak flop rate of our FFTs to be 264 Gflops/s. Each FFT requires 3.82 GFlops, so all 1.2M FFTs will require 17K seconds or 4.82 hours. This is obviously infeasible for clinical use.

Both the large working set size and the aggressive clinical deadline suggest we leverage distributed-memory machines. Another back-of-the-envelope calculation reveals their effectiveness. If we assign one processor per timepoint, then we utilize 750 processors performing 1,600 FFTs apiece. This requires 23 seconds in total and is well within the range of clinical feasibility. Of course, a real reconstruction performs more operations than FFTs and incurs some coordination overhead; the purpose of this chapter is to show that these additional considerations are not fatal, and to demonstrate empirically that it is possible to scale reconstruction codes up to massively parallel systems while meeting clinical deadlines.

7.2 Problem Description

We reconstruct the UWUTE dataset using an approach similar to that of Zhang, Pauley, and Levesque [62]. We formulate reconstruction as an optimization problem with a linear data consistency operator A, the SENSE operator, and a regularization term, $L(\cdot)$, to enforce sparsity in the reconstructed image. Thus, we seek the solution to:

$$\min_x \|Ax - y\|_2^2 + \alpha L(x).$$

Dimension	Description	Size	Operations
T	Timepoints	50	SVDs, GEMMs, Alltoallvs
C	Coil (Receive Channel)	8	Reduce_scatters, Allgathervs
PS	K-Space Data (Phase-shift)	100	SpMMs
RO	K-Space Data (Read out)	654	SpMMs
X	Spatial Dimension 2	208	SpMMs, FFTs
Y	Spatial Dimension 1	308	SpMMs, FFTs
Z	Spatial Dimension 0	480	SpMMs, FFTs

Table 7.1: Description of Dimensions in Non-Cartesian MR Image Reconstruction

Name	Description	Shape	Datatype	Size
`data`	Acquired K-space Data	$T \times C \times PS \times RO \times 1$	`float`	3.14 GB
`traj`	Sampling Trajectory	$T \times PS \times RO \times 3$	`float`	1.18 GB
`dcf`	Density Compensation Factor	$T \times PS \times RO \times 1$	`float`	0.38 GB
`maps`	Sensitivity Maps	$C \times X \times Y \times Z$	`complex`	1.97 GB
`img`	Reconstructed Image	$T \times X \times Y \times Z$	`complex`	185 GB

Table 7.2: Data Arrays in Non-Cartesian MR Image Reconstruction.

The data from the scanner is provided as a multi-dimensional array of single-precision complex floating point values which we denote the `data` array. Also provided as input are the sensitivity maps (denoted as `maps`) and the sampling trajectory (`trajectory`). The goal of the reconstruction is to produce an array `image` representing the 3D images at each timepoint. The exact dimensions of each array and their corresponding sizes are given in Table 7.2.

Data Consistency

We construct the SENSE linear operator A as explained in detail in Section 6.2. We order the multidimensional arrays with dimensions as given in Table 7.2, with the spatial dimensions varying the fastest. Under this arrangement, the SENSE operator matrix is block diagonal, with a block corresponding to each timepoint. Consequently, the solution for each timepoint is decoupled from all others, suggesting that we can apply the data consistency operator in parallel across timepoints without communication. We will exploit the decoupled nature of the SENSE operator when choosing a data decomposition, described later in Section 7.3.

Regularization

MRI researchers often seek to apply a regularization operation to bias the solution toward properties of known-good images. Regularization can be performed in both spatial and temporal dimensions. For example, a positivity constraint can be applied to ensure that no solution proposes a negative amount of mass within a voxel (a condition that is physically impossible). More advanced spatial regularization techniques include the discrete cosine transform, wavelet transforms, and spatial finite differences [47].

When collecting images over time (commonly known as "video"), it is desirable to maintain temporal coherence—the property that adjacent images in time appear nearly identical save for intrinsic differences. Due to the under-constrained and time-decoupled nature of the reconstruction problem, independent solvers working on adjacent images can arrive at different images that, when played as a video, produce a flickering effect due to their lack of temporal coherence. The flickering is neither pleasant to watch nor physically accurate, hence techniques are employed to minimize it. It is possible to use regularization in the time dimension to bias the solution toward images that are still consistent with the acquired data, but have sufficient temporal coherence to achieve diagnostic quality.

We employ the Locally Low-Rank regularization method (LLR) [62]. LLR is a generalization of the Globally Low-Rank method (GLR), which we describe first. A GLR regularization arranges a series of images into a "Casorati" matrix in which each matrix row corresponds to a location in space (voxel), and each matrix column corresponds to a particular point in time (image). Reading down a row of the Casorati matrix reveals how a particular voxel evolves over the course of the acquisition. We wish for this evolution to be smooth yet still represent the intrinsic motion in the image. A GLR regularization enforces this property by computing the singular value decomposition of the Casorati matrix, thresholding its singular values, and performing matrix multiplication to rebuild the matrix. From there, the reverse Casorati transformation can be applied to recover the regularized images.

The Locally Low-Rank regularization method differs from the Globally Low-Rank method in that LLR divides the images into small blocks, typically 8^2 or 8^3 in size, or generally b^d in size for d-dimensional images and performs a low-rank regularization on each of these small blocks. The LLR method can create imaging artifacts at block boundaries since adjacent blocks might be regularized differently. One technique for lessening these boundary effects is to regularize all overlapping blocks. Since the number of overlapping blocks is proportional to the number of voxels, this approach requires b^d more regularizations. Further adding to the cost of overlapping-blocks regularization is a dependence that induces a serialization in how the regularizations can be computed. Consider the perspective of a single voxel—it must participate in b^d block regularizations, i.e. the number of overlapping blocks in which it resides. One cannot perform all of these regularizations in parallel because it's not known how to merge the results. Instead, all regularizations for a particular voxel must proceed sequentially.

A compromise between regularizing non-overlapping blocks and overlapping blocks arises in the context of an iterative solver. We recognize that we apply the regularizer many times during a reconstruction, specifically once per iteration. Thus, we still choose to regularize only non-overlapping blocks, but we randomly shift the boundary of the blocks on every regularization. To implement this in distributed-memory, we compute and communicate a random d-dimensional offset vector $\hat{b}$ such that $0 \leq b_i < b, b_i \in \mathbb{I} \forall i$. Consequently, over the course of many iterations, the boundary effects of single-step regularizations are distributed across the entire image.

Iterative Solver

We employ the FISTA iterative solver [5]. FISTA is gradient method that seeks to minimize the sum of two convex functions f and g; in our case, f is the gradient operator, built on the data-consistency SENSE operator, and g is the regularization operator. A parameter λ controls the relative importance of each function in the summation. In every FISTA iteration, f and g are evaluated once. FISTA is a refinement of ISTA with a better step-size calculation.

7.3 A Distributed-memory Implementation

So far, we have presented a reconstruction problem that requires distributed-memory computing to be viable. In this section, we discuss the challenges that arise when porting the reconstruction code to a distributed-memory machine, present our strategy for doing so, and argue that it is a reasonable approach. We seek an implementation with several desirable properties:

Strong Scaling The code should enable good *strong scaling* performance. This metric assesses the performance of the code as increasing resources are used to solve a fixed-size problem instance. As an example, we expect that, under perfect strong scaling, doubling the size of the machine (i.e. number of processors) will halve the running time of a particular problem instance. Strong scaling is an important metric because it represents the code's ability to reduce running time arbitrarily. A code with perfect strong scaling will be able to meet any application deadline assuming unbounded computing resources. Furthermore, since the problem size is fixed by the scan procedure, the only way to reduce reconstruction time is by strong scaling.

Weak Scaling An ideal code should also have good *weak scaling* performance. Weak scaling assesses the performance of the code as the problem size and machine size increase proportionally. Weak scaling demonstrates the code's ability to handle increasingly larger problems. For example, we expect a reconstruction task with 10 timepoints and 10 processors to finish in the same time as a task with 20 timepoints and 20 processors. As scanners employ more receive channels, we can expect weak scaling to be important as problem sizes grow.

Correctness Lastly, the code should be *correct* in that it computes the same answer as a serial implementation regardless of the number of processors used to compute the solution. While correctness may seem to be obviously desirable, it is common practice in distributed-memory software development to make slight algorithmic changes that complement the underlying architecture. For example, the desire to avoid communication in distributed-memory convex optimization partially motivated the development of the ADMM algorithm [12]. Other distributed memory algorithms like Hogwild [57] relax correctness constraints and opt to compute the solution probabilistically instead of exactly. We desire to scale the MRI reconstruction code faithfully, since the approach 1) has been validated by medical researchers and 2) is still likely to run reasonably well.

Full bitwise equality is difficult to achieve due to the non-associativity of floating point arithmetic and the potential for collective arithmetic subroutines to re-order operations for better performance. Thus, we assume a slightly relaxed correctness metric—we expect distributed-memory results to be within machine epsilon of a serial result. This can be checked by computing the infinity norm of the difference between the actual result, x, and the expected result, x':

$$\|x' - x\|_\infty = \max{}_i |x'_i - x_i| < \epsilon.$$

Sources of Parallelism in MR Image Reconstruction

Substantial parallelism exists within the reconstruction problem. Here, we catalog all sources of parallelism in anticipation of mapping them to a distribute-memory architecture.

For our chosen vectorization of the image arrays, the SENSE operator is block diagonal, with blocks corresponding to each timepoint. This suggests that each block can be applied in parallel without communication, or generally that the SENSE operator is parallel across the time dimension. Similarly, the non-uniform Fourier Transforms (NUFFTs) within each receive channel can be computed in parallel—the NUFFTs are parallel across the channel dimension. With each NUFFT, a standard FFT operates on the three spatial dimensions. FFTs can be implemented recursively with independent tasks for computing subproblems, followed by a synchronized task to merge the results [25].

Parallelism also abounds in the LLR regularization step. Each Casorati block can be regularized independently, and within each regularization, the SVD and matrix multiplication contain opportunities for parallelism.

Sources of Locality in MR Image Reconstruction

We also consider data locality—the ability to re-use spatially or temporally nearby data—in our design. Locality is especially important because MRI reconstruction is limited by memory-bandwidth, so it's worthwhile to reduce memory traffic by exploiting locality. There are two main opportunities to leverage locality in our code:

- The same set of sensitivity maps can be shared by all timepoints.
- The same gridding matrix can be shared by all channel-images corresponding to the same timepoint.

Data Layout

Given the above opportunities for parallelism and locality, we must now decide how to distribute the problem instance across a set of processors. We make a few observations that suggest a particularly compelling layout:

- The most computationally complex operators are the SVDs and GEMMs found in an LLR block regularization. However, they are typically small: with 750 timepoints and a LLR blocksize of 8^3, each operates on a matrix of size 512-by-750. Furthermore, they're numerous: regularizing the entire image requires around 30M block regularizations, each comprising an SVD and GEMM. This large number of modest tasks suggests that we perform SVDs and GEMMs in shared memory, and scatter the full set of block regularization tasks across the machine. Alternatively, we can compute SVDs and GEMMs redundantly to reduce overall communication costs.

- The next most computationally complex operators are the 3D FFTs found in the SENSE operator. Our SENSE operator performs one forward and one inverse 3D FFT for each channel and for each timepoint, or 12K FFTs in total. The FFTs are small enough to fit into the memory of individual nodes of the machine, but not the caches, suggesting that we leverage shared-memory parallelism within each FFT distribute the set of FFT tasks across the machine.

Thus, we divide the processors into a grid with the same number of dimensions as the acquired data. We populate the time and coil dimensions with processors, and leave the remaining dimensions unit-length. On the UWUTE dataset, this approach induces a two-dimensional processor grid of shape P_t by P_c, with P_t processors in the time dimension and P_c in the channel dimension. The total number of processors is thus $P = P_t \cdot P_c$. We illustrate this arrangement for a processor grid with 3 channels and 5 timepoints in Figure 7.1.

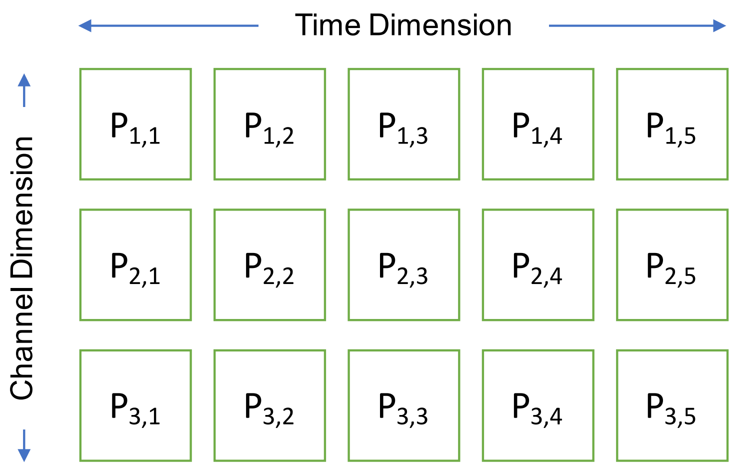


Figure 7.1: Proposed Processor Layout for Distributed-Memory MR Image Reconstruction

Onto the two dimensional processor grid we distribute the relevant multi-dimensional arrays in a two-dimensional blocked layout. If the number of array elements doesn't evenly divide the number of processors in a given dimension, we allow the last processor to have a slightly smaller block. If an array is not scattered across a dimension, we replicate its contents along that dimension. Figure 7.2 illustrates our data decomposition.

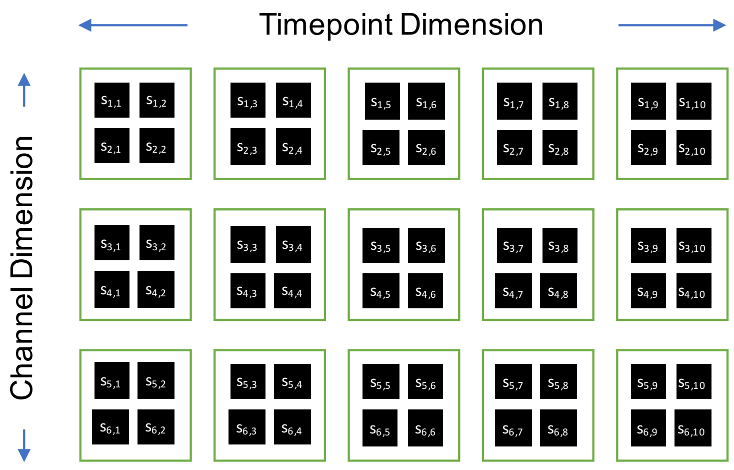

Timepoint Dimension
Channel Dimension
$s_{1,1}$ $s_{1,2}$ $s_{2,1}$ $s_{2,2}$
$s_{1,3}$ $s_{1,4}$ $s_{2,3}$ $s_{2,4}$
$s_{1,5}$ $s_{1,6}$ $s_{2,5}$ $s_{2,6}$
$s_{1,7}$ $s_{1,8}$ $s_{2,7}$ $s_{2,8}$
$s_{1,9}$ $s_{1,10}$ $s_{2,9}$ $s_{2,10}$
$s_{3,1}$ $s_{3,2}$ $s_{4,1}$ $s_{4,2}$
$s_{3,3}$ $s_{3,4}$ $s_{4,3}$ $s_{4,4}$
$s_{3,5}$ $s_{3,6}$ $s_{4,5}$ $s_{4,6}$
$s_{3,7}$ $s_{3,8}$ $s_{4,7}$ $s_{4,8}$
$s_{3,9}$ $s_{3,10}$ $s_{4,9}$ $s_{4,10}$
$s_{5,1}$ $s_{5,2}$ $s_{6,1}$ $s_{6,2}$
$s_{5,3}$ $s_{5,4}$ $s_{6,3}$ $s_{6,4}$
$s_{5,5}$ $s_{5,6}$ $s_{6,5}$ $s_{6,6}$
$s_{5,7}$ $s_{5,8}$ $s_{6,7}$ $s_{6,8}$
$s_{5,9}$ $s_{5,10}$ $s_{6,9}$ $s_{6,10}$

Figure 7.2: Distributed-Memory PICS (PICS-DM) divides the data across both the time and channel dimensions of the processor grid. It assigns blocks of "channel-images", the atomic unit of distributed-memory parallelism, to each processor according to that processor's location in the grid. Here, we show a 15 processor machine divided into a 5-by-3 grid. 10 timepoints, each with 6 channel-images, are distributed across this grid. Green squares represent processors (as in Figure 7.1), and black squares channel-images. The channel image $s_{x,y}$ corresponds to the image from the x-th channel and the y-th timepoint.

7.4 Distributed-Memory Algorithmic Extensions

Given the processor grid and data decomposition described in Section 7.3, we can now extend the reconstruction algorithms given in Section 6.2 to the distribute-memory setting.

Distributed-Memory FISTA

The FISTA routine has three major steps: apply the linear operator, apply the regularization operator, and perform scalar and vector-vector "bookkeeping" arithmetic to compute step sizes, gradients, and other search metadata. We cover the linear operator and regularization routines in the next section because they have non-trivial costs.

The bookkeeping arithmetic operates on vectors spanning the time dimension. Because the vectors are identically distributed, and the arithmetic can be reduced to embarrassingly-parallel AXBPY operations, the only distributed-memory communication necessary to maintain the FISTA search metadata is synchronization barriers before and after the AXPBYs. In practice, these barriers can be removed because the subsequent synchronous collectives in the data consistency and regularization operators enforce the synchronization. Thus, the FISTA routine requires no modification to target distributed-memory systems.

Distributed-Memory SENSE

The SENSE operator requires two slight modifications for distributed-memory systems.

Before explaining them, we distinguish between the *global* SENSE operator and the *local* SENSE operator. The global operator is the single operator being evaluated across the entire machine; it coordinates the evaluation of local SENSE operators on each node and performs the communication necessary to maintain correctness. The local operators are implemented with Indigo, hence our task reduces to inserting communication operations at strategic locations.

Since the SENSE operator is embarrassingly parallel across timepoints, we need only surround it with barriers to ensure consistency. As in the distributed-memory extensions to FISTA, these barriers are redundant and can be removed in the final implementation.

Parallelizing the global SENSE operator across the channel dimension is more challenging because each local SENSE operator must read from, and later accumulate into, the image that is potentially distributed across a subset of processors. We implement this by adding collective communication operations in the forward and adjoint SENSE routines. In the forward operator, named Sense-Forward-DM, an all-gather routine retrieves the portions of the image, appends them into one complete image, and broadcasts them to all processors working on the same time.

Algorithm 1 Sense-Forward-DM

```
1: **Input:** image array, distributed so subset $x_{t,c}$ belonging to processor $P_{t,c}$
2: **Output:** data array, distributed so subset $y_{t,c}$ belongs to processor $P_{t,c}$
3: for all processors $P_{t,c}$ in parallel do
4:     $x_t \leftarrow \textsc{Allgather}(x_{t,c}, \text{CHANNEL\_DIM})$
5:     $y_{t,c} \leftarrow \textsc{Local-Sense-Forward}(x_t)$
6: end for
```

Similarly, we add a reduce-scatter routine to the adjoint operator, Sense-Adjoint-DM. It collects the channel-images, sums them into a final image, and scatters it back across the team of processors. The Sense-Adjoint-DM algorithm is given in Algorithm 2.

Algorithm 2 Sense-Adjoint-DM

```
1: Input: data array, distributed so subset $y_{t,c}$ belongs to processor $P_{t,c}$
2: Output: image array, distributed so subset $x_{t,c}$ belonging to processor $P_{t,c}$
3: for all processors $P_{t,c}$ in parallel do
4:     $x_{t,c} \leftarrow$ LOCAL-SENSE-ADJOINT($y_{t,c}$)
5:     $x_{t,c} \leftarrow$ REDUCE-SCATTER($x_{t,c}$, CHANNEL_DIM)
6: end for
```

7.5 Distributed-Memory LLR

Extending Locally Low-Rank regularization to the distributed memory setting is nontrivial because it requires global communication between all pairs of processors. The challenge arises because the regularization step operates across the time dimension and possesses parallelism in the spatial dimensions, while the SENSE operator operates across the spatial dimensions and possesses parallelism in the time dimension. To implement both SENSE and LLR routines, we must perform global transpose-like operations, or `Alltoall`s in MPI parlance, to move data back and forth between temporal and spatial decompositions every iteration. `Alltoall`s are typically expensive because they require every processor send data to every other processor in the machine—they can tax the machine's interconnect network.

In Algorithm 3, we present the LLR regularization routine augmented for distributed-memory systems. It takes as input a multidimensional array representing a series of images; the array has one time dimension and multiple spatial dimensions. The array is distributed across our 2D processor grid with timepoints distributed block-wise across the processor grid's time dimension, and the spatial dimensions distributed block-wise across the other processor grid's channel dimension. This layout is the output of a SENSE-DM operation.

LLR-DM proceeds by randomly choosing a shift factor along which the Casorati blocks will be selected. It then communicates the shift factor to all processors in the machine so they perform the same shifts. Next, each processor packages its local data into blocks and performs an `Alltoall` along the processor grid's time dimension to transpose from the temporal decomposition into a spatial one. Upon return from the `Alltoall`, each processor has the data necessary to regularize its subset of the Casorati blocks. Once regularized, a second `Alltoall` restores the temporal decomposition and the global regularization step is complete. In this way, each processor performs a roughly equal share of the total block-regularizations.

7.6 Communication Pattern

Understanding the distributed-memory extensions to the SENSE operator, the Locally Low-Rank (LLR) regularization operator, and the FISTA solver allows us to describe the overall distributed-memory communication pattern in a PICS-DM reconstruction. The iteration

Algorithm 3 LLR-DM

```
1: **Input:** multidimensional image array $x$ distributed across 2D processor grid, with local
   portion $x_{t,c}$.
2: **Input:** LLR block size $b$
3: **Output:** $x'$, an regularized image with the same layout as $x$.
4: **for** all processors $P_{t,c}$ in parallel **do**
5:     $\hat{s} \leftarrow$ GenerateRandomShiftVector($len = 3, max = b$)
6:     $\hat{s} \leftarrow Broadcast(\hat{s})$
7:     $z \leftarrow$ FormCasoratiBlocks($x_{t,c}, b, \hat{s}$)                ▷ local transpose
8:     $z \leftarrow$ Alltoallv($z$, TIME_DIM)
9:     $z \leftarrow$ LLR-Local($z$)                                          ▷ SVDs, GEMMs
10:    $z \leftarrow$ Alltoallv($z$, TIME_DIM)
11:    $x_{t,c} \leftarrow$ UndoCasoratiBlocks($z, b, \hat{s}$)               ▷ local transpose
12: **end for**
```

begins with the image x block-distributed across a two-dimensional processor grid. The processors are group into teams that work on common subsets of timepoints ("timepoint teams") and teams that work on common subsets of channels ("channel teams"). The iteration proceeds as follows:

1. Processors perform an all-gather operation within the timepoint teams. Subsequently, all processors within a timepoint team have copies of the team's assigned subset of images.

2. Processors apply a local SENSE operator, generating channel-images.

3. Processors perform a reduce-scatter operation within the timepoint team. This operation sums the channel-images and scatters the result back across the timepoint team, restoring the original layout.

4. The lead processor computes a shift-vector to determine LLR regularization boundaries, and broadcasts it to all other processors.

5. Processors perform an all-to-all operation to transpose from a temporal decomposition to a spatial one.

6. Processors perform local LLR block regularizations.

7. Processors perform a second all-to-all operation to restore the spatial decomposition.

8. The iteration is complete.

This communication pattern is illustrated in Figure 7.3. Along the way, the iterative solver performs local vector-vector arithmetic to maintain search metadata. This results in no distributed-memory communication.

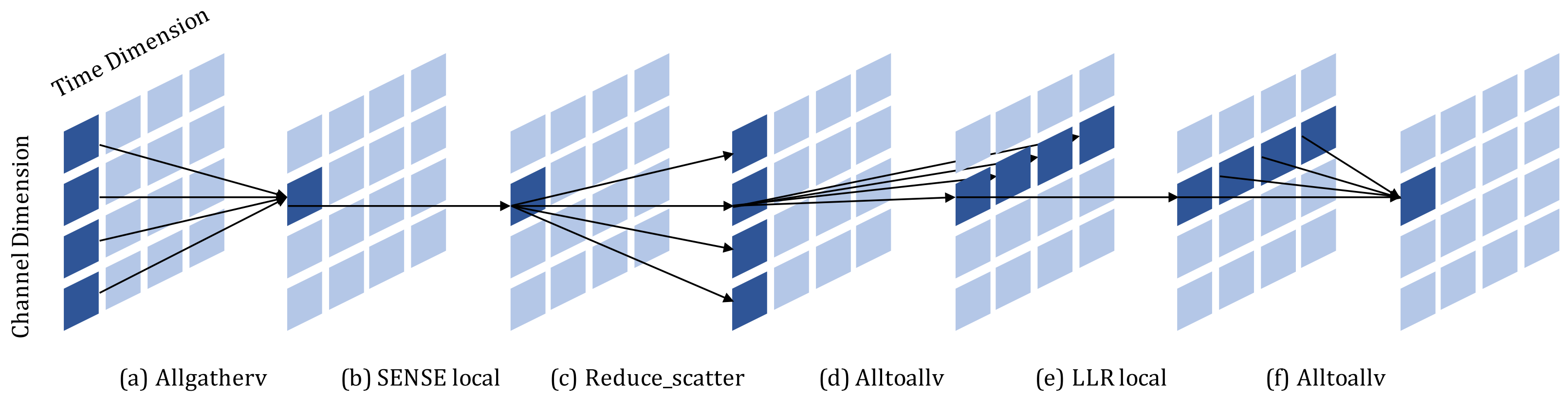


Figure 7.3: Communication operations from the perspective of a single processor (here, $P_1, 2$) that occur during one iteration. The first three operations implement the distributed-memory SENSE operator (Section 7.4): (a) an all-gather collects fragments of the image distributed in the channel dimension, next (b) processors perform a local SENSE operation, then the resulting images are summed and scattered in the `Reduce_scatter` step. The last three operations implement the distributed-memory LLR regularization (Section 7.5): (c) an all-to-all transposes the data from a temporal decomposition to a spatial decomposition, (d) a local routine constructs, regularizes, and deconstructs each Casorati block, and (e) another all-to-all restores the original, time-decomposed layout. The rapid switching between the time- and space-centric layouts makes distributed-memory MR image reconstruction challenging from a parallel computing perspective.

7.7 Performance Results

We demonstrate the viability of the distributed-memory approach through experiments on a modern supercomputer. In this section, we give results indicating that our code makes efficient use of distributed-memory systems for the UWUTE1 dataset.

Test Setup

We evaluated our code on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). Cori is a Cray XC40 with a peak performance of 30 petaflops. It is composed of two partitions that differ in their processor architectures.

The Haswell partition contains 2,388 nodes, each with a 32-core Intel Xeon E5-2698 v3 processors and 128 Gigabytes (GB) of double data-rate fourth generation (DDR4) memory. Each node is capable of 1.2 teraflops of compute performance and 120 GB/sec of sustained memory bandwidth.

The Knights Landing (KNL) partition contains 9,688 nodes, each with a 68-core Intel Xeon Phi 7250 KNL processor, 96 GB of DDR4 memory, and 16 GB of multi-channel dynamic random access memory (MCDRAM). Each KNL node is capable of 3 teraflops per node, 102 GB/sec of dynamic random access memory (DRAM) STREAM bandwidth, and 460 GB/sec of MCDRAM STREAM bandwith. We operate the MCDRAM in the "cache memory mode' where the MCDRAM serves as a cache on main memory.

Cori employs the Cray "Dragonfly" interconnect. In Cori, four nodes form a blade with a shared router. Sixteen blades form a chassis with all-to-all circuit board connections. Chassis are connected with copper cables to form cabinets, and cabinets are connected with optical cables to form the top-level network. The interconnect offers 45 Terabytes (TB)/sec of bisection-bandwidth—the total bandwidth attainable when half of the processors send data to the other half.

We launch jobs using the Slurm batch system, which reserves a group of processor nodes within the machine. Because we reserve far fewer nodes than Cori possesses, and because many other jobs are running simultaneously, there is no guarantee that our nodes have exclusive use of the network links between them. Consequently, other jobs on the machine may create network traffic that influences the performance of our code. To lessen these effects, we perform our experiments ten times and report the average result.

Weak Scaling Performance Results

First, we seek to understand how our code behaves as problem sizes increase. In the distributed-memory setting, it is most interesting to assess size increases in dimensions that are distributed across the processor grid—the time and channel dimensions. Channel dimensions are not expected to increase dramatically because additional channels achieve physics-induced diminishing returns, so we focus on the time dimension. The number of timepoints

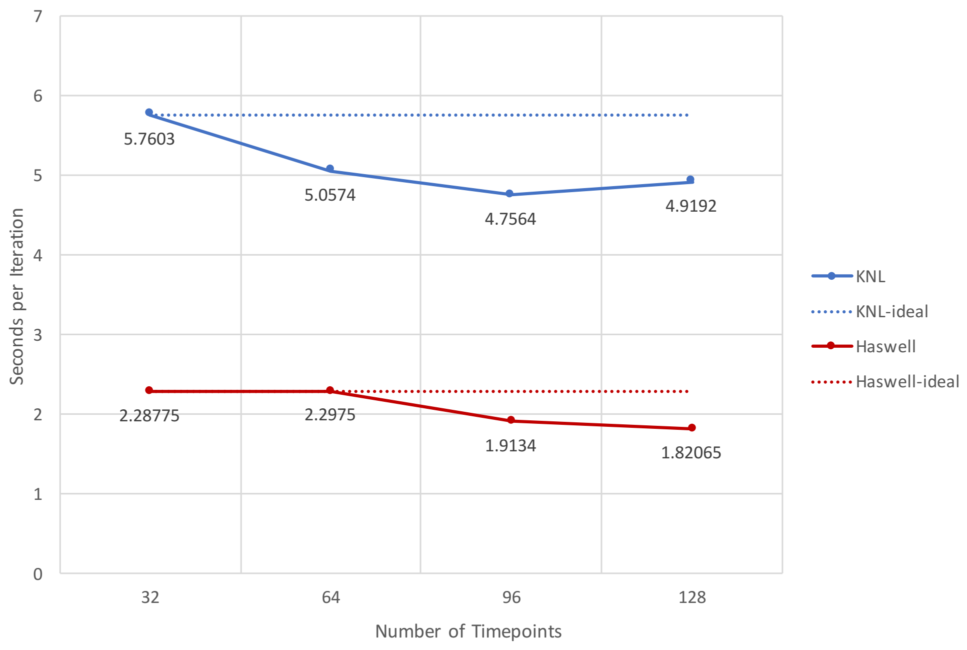


Figure 7.4: Weak-scaling performance of the distributed-memory PICS implementation on Cori's KNL and Haswell processors. The machine size is proportional to the number of timepoints. The relatively flat trends indicate that PICS-DM can scale to the largest problem we encountered without loss of efficiency.

in an acquisition grows as a function of two parameters, the total number of data points acquire and the grouping of those points into distinct timepoints.

We designed an experiment to measure code performance as a function of problem size. We perform a series of reconstructions on the University of Wisconsin Ultra-short Time Echo (UWUTE) dataset in which we artificially crop the length of the time dimension to the desired problem size. As the problem size grows, we grow the machine size correspondingly so that we maintain a constant amount of work per processor. This comprises a "weak scaling" benchmark.

We evaluate the PICS-DM code on reconstruction problems ranging in size from 32 timepoints to 128. We assigned one processor per channel-timepoint; thus with 8 acquisition channels the machine ranged in size from 256 to 1,024 nodes. We measured the time to run FISTA for 50 iterations and averaged the iteration times. We plot the results in Figure 7.4.

Figure 7.4 shows perfect weak scaling behavior—the time per iteration remains constant as the problem size and machine size increase proportionally. In fact, overall performance improves as much as 20% as the problem size increases. Some of this difference is likely attributable to variability in network performance on the shared machine.

The perfect weak scaling result suggests that the challenge of reconstructing large time-series images is not a technical problem but merely an economic one. PICS-DM can utilize additional processors without a loss of parallel efficiency so practitioners are able provision just enough computational resources to produce solutions within their given deadline. Furthermore, the result indicates that end of the perfect scaling regime is beyond the problem sizes tested. It is possible that the scaling regime will continue for future, larger problems.

Strong Scaling Performance Results

Our second main experiment seeks to understand the behavior of PICS-DM on a particular problem instance as machine sizes increase. The computational power of the largest supercomputers has increased exponentially over the last three decades [63], and the top machines current offer 100 petaFLOPs of peak performance. While our image reconstruction problems require substantially less computational power, they do benefit from increases in processing power (FLOP/s) per unit area (at both transistor and machine scale) and per unit energy.

We performed an experiment in which we reconstructed a fixed-size problem on increasingly larger machines, i.e. a strong scaling experiment. This metric measures a code's ability to efficiently utilize additional computational resources. For this experiment, we cropped the UWUTE dataset to 128 timepoints and 8 channels, and mapped it onto machines ranging in size from 32 to 128 nodes. In each case, the processor grid used four teams across the channel dimension ($P_c = 4$). We measured the average iteration time of 50 iterations and plot it in Figure 7.5.

The strong-scaling data in Figure 7.5 show parallel efficiency that is slightly better than perfect—doubling the machine size results in more than two-fold improvements in iteration time. This counter-intuitive result is likely the consequence of the per-node working sets fitting in faster levels of the memory hierarchy. When the problem instance is distributed across more nodes, each node has a smaller worker set and it's more likely to fit in cache.

From an application perspective, the results indicate that reconstruction latency can be improved by provisioning more computational resources, and the additional resources will deliver a corresponding amount of performance. As in the weak-scaling experiment, the scaling regime holds throughout the range tested. Meeting a given reconstruction deadline is consequently no longer a technical challenge, but an economic one.

Image Results and Discussion

A coronal slice of the reconstructed UWUTE dataset is shown in Figure 7.6. The full reconstruction produced 64 images covering the breathing cycle of the exam subject.

Parallel Parameter Search One drawback of iterative, LLR-regularized reconstruction is that some scalar parameters must be chosen to produce images of diagnostic quality. These parameters include the iterative solver's λ, which balances data consistency and regularization, and the low-rank regularizer's α, which limits the amount of motion in the image. Practitioners must often try multiple values for these parameters until the reconstruction

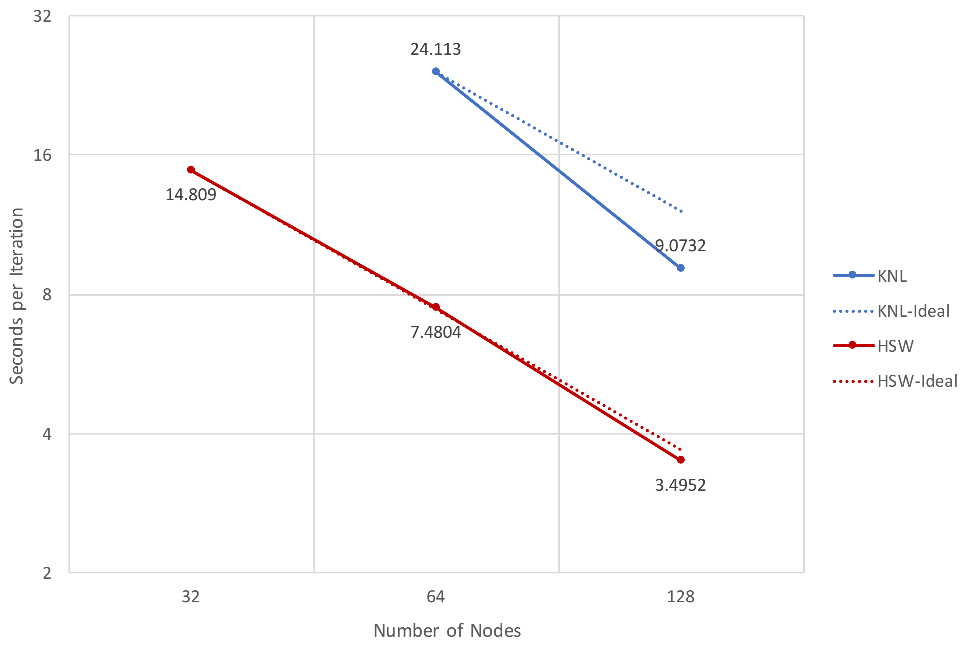


Figure 7.5: Strong-scaling performance of the distributed-memory PICS implementation on the Cori supercomputer. The problem size was fixed at 128 timepoints, 8 receive channels, and the standard UWUTE1 spatial dimensions. Both the KNL and Haswell series achieve superlinear strong scaling, indicating that the code is making efficient use of the distributed-memory machine. From an application perspective, these results suggest that practitioners can justifiably provision more computational resources to meet satisfy aggressive reconstruction deadlines.

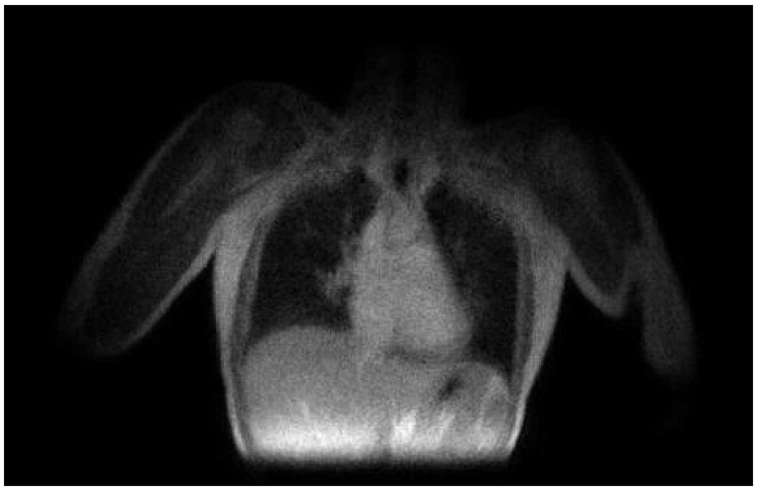
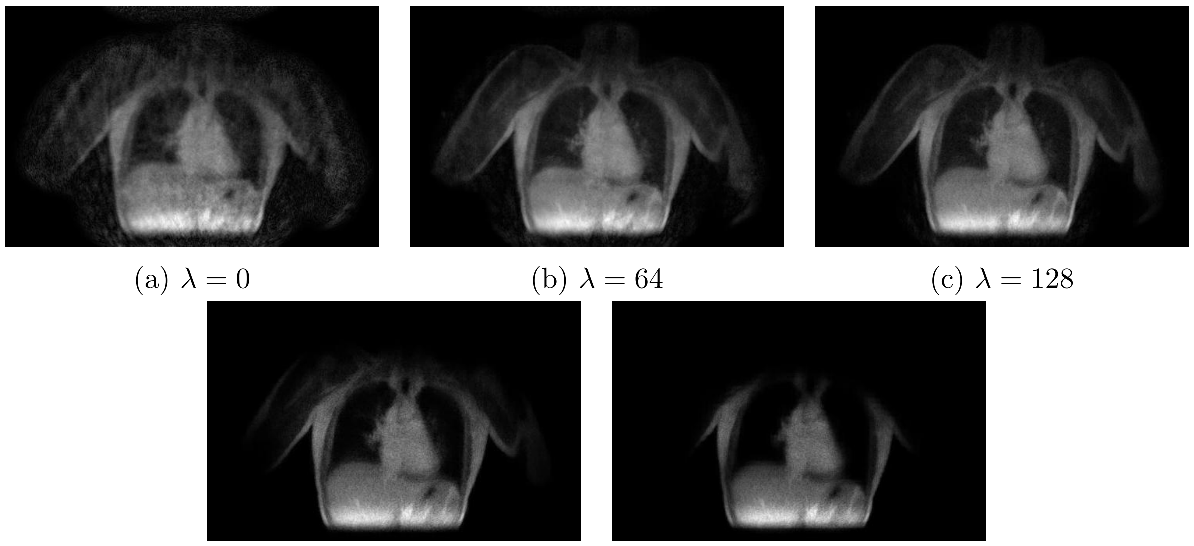


Figure 7.6: Example coronal image of a human thorax reconstructed from the UWUTE dataset. The entire dataset was reconstructed in under three minutes.

code produces acceptable images. Figure 7.7 shows reconstructed images for multiple values of λ.



(a) $\lambda = 0$ (b) $\lambda = 64$ (c) $\lambda = 128$

(d) $\lambda = 512$ (e) $\lambda = 1024$

Figure 7.7: In regularized reconstructions, it is necessary to choose a value λ that balances consistency with the acquired data with the effect of the regularization method. These images show reconstructions with various values of λ. They are coronal slices taken from a four-dimensional reconstruction of the UWUTE dataset.

Without parallel computing, tuning for good values of λ and α requires iteratively performing reconstructions with difference parameters. However, given sufficient computing resources, it is possible to performing reconstructions in parallel since each reconstruction is independent of others.

7.8 Future Work

There are three general directions that we envision for continuing this work.

MRI Performance Improvements

Additional opportunities exist for improving the performance of the PICS-DM code should it be required in the future. They are best explained in terms of shortcomings of the current implementation.

First, the PICS-DM decomposition strategy identifies the "channel-image" as the atomic unit of distributed-memory parallelism, i.e. a channel-image cannot be distributed across multiple distributed-memory nodes. This choice limits the number of nodes that can participate in a reconstruction to the total number of channel-images in the dataset. For the UWUTE dataset, there are 750 timepoints and 8 channels, or 6,000 channel-images; PICS-DM would not be able to scale to Cori, which has 12,076 nodes, without further parallel decomposition. Fortunately, FFT- and stencil-style parallelism exists in the spatial dimensions—one or more spatial dimensions could be distributed to enable all 12,076 nodes to participate in the reconstruction, at the cost of substantial code complexity.

Cloud-based Image Reconstruction

Cloud computing is an attractive platform for MRI researchers and clinicians because of it's accessibility and scalability. Attaining performance for tightly-coupled distributed-memory applications on the cloud can be challenging for several reasons. First, the interconnect networks typically employ Transfer Control Protocol (TCP) over Ethernet, rather than the custom high-performance hardware and protocols found in supercomputers like Cori. Second, cloud providers typically over-provision machines, which can substantially reduce performance. And lastly, cloud providers do no provide efficient batch scheduling for large jobs. Nevertheless, we are optimistic that PICS-DM can obtain good performance on the cloud because the dominant hurdle—communication—comprises a small portion of the overall runtime.

We experimented with running PICS-DM on the Amazon cloud using StarCluster [48], a tool that provisions and configures an MPI cluster on demand. We succeeded in running on virtual clusters composed of multi-core CPU instances, but we were unable to run on GPU clusters because of software configuration problems. Future work is required to fix such problems and achieve portability across cloud platforms.

Beyond MRI: General Image Reconstruction

Although PICS-DM is specific to MR image reconstruction, other image reconstruction problems exist that would benefit from distributed-memory implementations. To address their demand, we envision a multidimensional array library, compatible with Indigo, that implements features required for image reconstruction. In particular, MRI requires a means of specifying the data distribution/blocking scheme, as well as communication primitives for reductions and transposes within prescribed dimensions. There has been substantial work on distributed-memory array abstractions [70, 33, 71], yet none were sophisticated enough to handle the transposes involved in LLR regularization. Even with a full-featured array library, distributed-memory computing remains challenging and we are pessimistic about automatic techniques for large-scale parallelization.

Chapter 8

Related Work

Our work builds on work in image reconstruction toolkits, domain-specific programming languages, sparse linear algebra, and high-performance medical imaging.

8.1 Image Reconstruction Toolkits

A variety of software packages exist that aid in solving image reconstruction problems. Like Indigo, they commonly employ a structured linear operator abstraction as the primary means of representing a linear imaging system.

- The Berkeley Advanced Reconstruction Toolkit (BART) [64] implements a superset of Indigo's operators on CPU platforms, and a limited set on GPU platforms. BART uses a direct griding method which computes blur coefficients on the fly, unlike Indigo which pre-computes them and stores them in explicit griding matrices. BART served as the primary reference implementation for the design of Indigo.
- The Michigan Image Reconstruction Toolbox (MIRT) [24] provides a `fatrix` type that represents a "fake matrix" and resembles Indigo's operator abstraction.
- The PyOP package [52] provides a linear operator abstraction on top of common operations from the Python software ecosystem. PyOP aims to leverage the lower time and space complexity of certain linear transformations by implementing them as matrix-free operators.

While the above reconstruction toolkits maintain a linear operator representation similar to Indigo's, none perform high-level transformations to improve evaluation performance.

Gadgetron [28] is another framework for image reconstruction, but it takes a different approach. Instead building on the linear operator abstraction, it provides a broad collection of "gadgets" that can be declaratively assembled into a network of operations that implements a desired reconstruction task. Gadgets typical represent higher-level operations than individual linear transformations, and extend beyond computational operations to input/output operations and data manipulation. Gadgetron provides gadgets for noise adjustment, casting between different numerical precisions, and reading and writing images. Reconstruction tasks are wholly contained in single gadgets. GPUs are supported by implementing separate gadgets, and explicitly invoking them in the user-level network description. The decision to use gadgets hinders portabliity, since gadgets must be re-implemented for each new platform, and optimization across gadgets isn't possible.

8.2 Domain-Specific Languages

Substantial work has shown that domain-specific languages are effective tools for achieving high performance. Examples include Halide for feed-forward image-processing pipelines [56], Simit for finite-element simulations [37], Pochoir for stencils [61], Asp for stencils [34], Spiral for digital signal processing [54], Liszt and Ebb for partial differential equations over mesh data structures [20, 7], and Opt for image optimization problems [21]. These languages share the technique of restricting generality to aid analysis and transformation, but do not cover the image reconstruction domain sufficiently to be useful.

Halide is especially relevant to our work because it implements broad fusion and reordering optimizations across operators. These optimizations are possible because Halide's stated domain, feed-forward image processing algorithms, is sufficiently regular that a transformation engine can reason about the correctness of possible transformations. The domain is surprisingly broad, covering stencils and FFTs simultaneously, and so seems a likely candidate for image reconstruction codes. However, the fundamentally sparse and irregular operators found in many of our case studies do not fall within this domain because of the necessary array-indirect access pattern they entail. These operators include the patch-selection operators from microscopy and magnetic particle imaging, and the gridding operators from non-Cartesian MRI reconstruction.

Domain-specific approaches have also been explored within the convex optimization community. CVX provides a framework for defining optimization problems which are convex by construction, and mathematically reduces the problem to target general solving routines [27]. CVXGEN generates C code that implements a custom quadratic program solver designed for use in embedded devices [45]. Chu et al. also explore code generation for convex problems [17], but focus primarily on correctness than performance. Cvxflow [69] and ProxImaL [29] propose a computation graph structure for convex problems and reorder the graph to improve performance. Our work is complementary to these techniques as it provides a methodology for cross-operator optimizations within the subset of operators that represent the full linear operator.

Our technique of separating the transformation recipe from the code to be transformed has been explored in other DSLs. We examine two here: CHiLL [3] and Halide [56]. In these DSLs, the recipes are procedural in nature, but differences arise in their expressiveness due to considerations from the application domain and intended platform(s). CHiLL uses transformation recipes to optimize scientific stencil computations. The CHiLL compiler ingests loop annotations that specify loop reordering transformations and application-level concepts like ghost zones. Halide's transformation recipes ("schedules" in Halide parlance) enable transformations to be applied to kernel objects in an image processing pipeline. Common Halide transformation include tiling, parallelization, and vectorization. Recent work on heuristics for scheduling Halide programs has been promising [49] and affirms the transformation recipe approach.

8.3 Sparse Linear Algebra

Indigo's collection of linear operators is similar to those provided by the SciPy [32] and Matlab [44]. Neither of these packages employ transformations to achieve better performance.

The inspector-executor model is commonly applied to sparse problems to select good storage formats or tuning parameters. OSKI [66] uses empirical performance tuning to select block sizes on a per-platform basis. Sparso [58] exploits optimization opportunities across sparse matrix operations, but its implementation in MKL yielded no benefit on our matrices. The CUDA sparse BLAS library [6] also performs minor inspection in construction of hybrid ELL+COO matrices. Our exclusive write property could be easily included in any of these inspector-executor libraries.

8.4 Fast Image Reconstruction

Efforts into high-performance image reconstruction can be classified as shared-memory techniques or distributed-memory techniques. In the shared-memory space, most efforts emply the traditional HPC approach of presenting ad-hoc implementations of particular reconstruction algorithms [36, 8, 55, 50, 15, 16].

Our explorations into distributed-memory reconstruction (Chapter 7) have similarities in other efforts, including:

- The Gadgetron package supports distributed-memory reconstruction on cloud and HPC systems. It uses a master-worker system architecture, where coarse-grain tasks are dispatched from a master node to workers running local Gadgetron instances, and TCP as the interconnect protocol. This architecture works well for reconstruction tasks with large, de-coupled subproblems, like independent reconstruction of planes in a volume. Performance would likely suffer, however, on tightly-coupled 3D reconstruction tasks because of the high communication latency induced by transport protocol, and the bottleneck that arises at the master node.
- The SHARP package [43] implements massively-parallel reconstruction for ptychography on GPU clusters. It is a high-performance implementation, but it is limited to ptychography and not portable to machines with other node architectures.

- Other packages for distributed-memory MRI reconstruction include PowerGrid [14], and Impatient MRI [68]. They are ad-hoc implementations of MRI reconstruction tasks and would have difficulty extending to other imaging modalities.

- Distributed-memory regularization is rarely explored because regularization workloads aren't computationally expensive enough to warrant massive parallelism. Some examples do exist, including a de-noising algorithm that runs across a cluster of GPUs [18].

Chapter 9

Conclusion

We have presented a domain-specific approach to the productive implementation of fast image reconstruction codes. The approach was manifest primarily in Indigo, a domain-specific language for image reconstruction on multi-core CPU and GPU machines. The primary insight that makes Indigo effective is its specific choice of domain abstraction for program entities—structured linear operators. This abstraction is rich enough that application developers can use it to build codes for a variety of reconstruction problems, but sufficiently restricted that program transformations can be applied automatically without complicated program analyses. It enables performance engineers to encode the transformations they would otherwise perform manually, in turn reducing the need for future performance engineering efforts.

The structured linear operator abstraction had some unexpected benefits. From a human perspective, it serves as a *lingua franca* for imaging scientists, who needn't be familiar with low-level programming, and performance engineers, who needn't be familiar with the application. The abstraction also aids portability because common linear operations—matrix multiplication, FFTs, etc.—are widely implemented, so the burden of porting Indigo to new platforms is dramatically reduced. Finally, the abstraction provided a framework for reasoning about program transformations, and even inspired transformations that weren't obvious in the original operator formulation.

The structure linear operator abstraction, and the domain-specific approach in general, is not without drawbacks. Indigo likely leaves some performance on the table for operators that don't possess a fast implementation. For example, the crop operator could be implemented directly with memory copies, but is instead represented as a sparse matrix with strategically-placed ones and evaluated via matrix-vector multiplication. This design decision is locally suboptimal, and the global optimality is a complicated function of the surrounding operators.

Another drawback is the semantic degradation that occurs as the application moves across the domain boundary. In the ptychography application (Section 6.4), a normalization term in a series of otherwise linear transformations placed the operator just outside the domain of Indigo. Despite the mismatch, we were still able to apply our techniques to the linear operators that bookended the normalization term, at the expense of additional application complexity and missed optimization opportunities. Including the normalization term in our domain abstraction, while retaining the power of Indigo's transformation engine, is a significant and open question for future work.

The performance results from the case studies (Chapter 6) indicate that Indigo can achieve a significant fraction of the Roofline peak. Furthermore, the performance is not merely a result of a static implementation—the program transformations applied by Indigo resulted in integer-factor speedups for the same operator on the same platform. These transformations might be performed manually by a performance programmer, but Indigo provides a framework for him or her to encode them to be applied automatically.

Chapter 7 utilized Indigo (a shared-memory code) in a distributed-memory setting. We gave a parallel algorithm suitable for locally low-rank regularized MR image reconstruction in distributed-memory. Performance results indicate that our code, `picsdm`, can perform 100 iterations on a hero-class problem in three minutes, placing it within the range of clinical feasibility. Furthermore, it achieves perfect strong scaling on up to 3,072 processor cores (a reasonable machine size for the problem at hand), suggesting that it can make efficient use of more computational resources to meet even more aggressive deadlines.

We believe our work demonstrates the value in utilizing a domain-specific language, and we hope that our techniques will inspire future languages in other domains.

9.1 Future Work

Our work suggests several directions for future research and development efforts. These include:

Expanding Indigo's Supported Domain

Indigo's domain abstraction covers linear transformations, but other, non-linear operations arise in imaging pipelines. One notable example is the normalization operation in the middle of the ptychography operator. Another example is the rectified linear units present in convolutional neural networks. It would be desirable to "grow" Indigo's domain to cover these operations so that operators in those fields could benefit from the program transformations that Indigo can apply.

We are pessimistic that general operators can be included in the domain, but we see two avenues for growing the domain. First, a likely next step is supporting affine transformations, which have the form

$$z = Ax + y.$$

This transformation can be expressed as a linear transformation between an augmented matrix and a higher-dimensional vector:

$$\begin{bmatrix} z \\ 1 \end{bmatrix} = \begin{bmatrix} A & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}$$

Support for affine transformations can benefit models of imaging systems that incorporate additive noise.

Second, it may be possible to expand the domain by tagging operators with properties relevant to the transformation engine. Currently in Indigo, all operators are implicitly tagged as linear operators. We could potentially allow operators to be tagged explicitly, with mathematical tags such as “linear” and “affine”, or operational tags such as “elementwise” and “in-place”. Such a tagging system would avoid the challenge of program analysis by deferring the task to programmers.

Both of these extensions to Indigo would substantially increase the number and complexity of possible transformations.

Automatic Generation of Transformation Recipes

Indigo currently requires programmers to specify a transformation recipe to be applied to an operator tree. This approach suffices for performance experts, since it gives them fine-grained control over the resultant tree, but it poses a challenge to application scientists who aren’t familiar with performance programming. It would be desirable to automatically generate good transformation recipes. This could happen via empirical search, but the search space is too large to search exhaustively, so heuristic methods are likely needed to guide a search routine or guess good recipes outright. Devising heuristics for optimizing Indigo ASTs is challenging, but solutions would aid adoption of Indigo.

Distributed-memory Image Reconstruction

In our distributed-memory extensions to Indigo (Chapter 7), the MRI scan data is store as a multidimensional array scattered across a multidimensional processor grid. In alternating between the SENSE operator that operates across spatial dimensions, and the low-rank regularization that operators across the time dimension, a number of non-trivial transposes, broadcasts, reduces, scatters, and gathers occur within the multidimensional array. Our implementation of these operation was ad-hoc in the sense that we make specific calls to low-level communication routines. A better long-term solution would be to implement a true multidimensional array object that supports distributed memory, such as DistNumpy [71], but with support for the operations common to imaging.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Reading, MA: Addison-Wesley, 1986.

[2] J. Ansel et al. "OpenTuner: An extensible framework for program autotuning". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT).* Aug. 2014, pp. 303–315. DOI: 10.1145/2628071.2628092.

[3] Protonu Basu et al. *Compiler Generation and Autotuning of Communication-Avoiding Operators for Geometric Multigrid.* Dec. 2013.

[4] P. J. Beatty, D. G. Nishimura, and J. M. Pauly. "Rapid gridding reconstruction with a minimal oversampling ratio". In: *IEEE Transactions on Medical Imaging* 24.6 (June 2005), pp. 799–808. ISSN: 0278-0062. DOI: 10.1109/TMI.2005.848376.

[5] A. Beck and M. Teboulle. "A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems". In: *SIAM Journal on Imaging Sciences* 2.1 (2009), pp. 183–202. DOI: 10.1137/080716542. eprint: https://doi.org/10.1137/080716542. URL: https://doi.org/10.1137/080716542.

[6] Nathan Bell and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, Dec. 2008.

[7] Gilbert Louis Bernstein et al. "Ebb: A DSL for Physical Simulation on CPUs and GPUs". In: *ACM Trans. Graph.* 35.2 (May 2016), 21:1–21:12. ISSN: 0730-0301. DOI: 10.1145/2892632. URL: http://doi.acm.org/10.1145/2892632.

[8] C. Bilen, Y. Wang, and I. W. Selesnick. "High-Speed Compressed Sensing Reconstruction in Dynamic Parallel MRI Using Augmented Lagrangian and Parallel Processing". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 2.3 (Sept. 2012), pp. 370–379. ISSN: 2156-3357. DOI: 10.1109/JETCAS.2012.2217032.

[9] Kai Tobias Block, Martin Uecker, and Jens Frahm. "Undersampled radial MRI with multiple coils. Iterative image reconstruction using a total variation constraint". In: *Magnetic Resonance in Medicine* 57.6 (2007), pp. 1086–1098. ISSN: 1522-2594. DOI: 10.1002/mrm.21236. URL: http://dx.doi.org/10.1002/mrm.21236.

[10] Uday Kumar Reddy Bondhugula. "Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model". AAI3325799. PhD thesis. Columbus, OH, USA, 2008. ISBN: 978-0-549-76796-1.

[11] Stephen Boyd et al. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Found. Trends Mach. Learn.* 3.1 (Jan. 2011), pp. 1–122. ISSN: 1935-8237. DOI: 10.1561/2200000016. URL: http://dx.doi.org/10.1561/2200000016.

[12] Stephen Boyd et al. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Found. Trends Mach. Learn.* 3.1 (Jan. 2011), pp. 1–122. ISSN: 1935-8237. DOI: 10.1561/2200000016. URL: http://dx.doi.org/10.1561/2200000016.

[13] R. W. Buccigrossi and E. P. Simoncelli. "Image compression via joint statistical characterization in the wavelet domain". In: *IEEE Transactions on Image Processing* 8.12 (Dec. 1999), pp. 1688–1701. ISSN: 1057-7149. DOI: 10.1109/83.806616.

[14] A. Cerjanic et al. "PowerGrid: A open source library for accelerated iterative magnetic resonance image reconstruction". In: *Proc. Intl. Soc. Mag. Res. Med.* 2016, p. 525. URL: http://indexsmart.mirasmart.com/ISMRM2016/PDFfiles/0525.html.

[15] C. H. Chang and J. Ji. "Compressed sensing MRI with multi-channel data using multicore processors". In: *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. Sept. 2009, pp. 2684–2687. DOI: 10.1109/IEMBS.2009.5334095.

[16] Xiangdong Yu Ching-Hua Chang and Jim X. Ji. "Compressed Sensing MRI Reconstruction from 3D Multichannel Data Using GPUs". In: *Magnetic Resonance in Medicine* (). DOI: 10.1002/mrm.26636.

[17] Eric Chu et al. "Code Generation for Embedded Second-Order Cone Programming". In: 2013.

[18] S. Cuomo, A. Galletti, and L. Marcellino. "A GPU Algorithm in a Distributed Computing System for 3D MRI Denoising". In: *2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*. Nov. 2015, pp. 557–562. DOI: 10.1109/3PGCIC.2015.77.

[19] I. Daubechies, M. Defrise, and C. De Mol. "An iterative thresholding algorithm for linear inverse problems with a sparsity constraint". In: *Communications on Pure and Applied Mathematics* 57.11 (2004), pp. 1413–1457. DOI: 10.1002/cpa.20042. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.20042.

[20] Zachary DeVito et al. "Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 9:1–9:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063396. URL: http://doi.acm.org/10.1145/2063384.2063396.

[21] Zachary DeVito et al. "Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging". In: *arXiv:1604.06525* (2016).

[22] Michael Driscoll. “Subdivision Surface Evaluation as Sparse Matrix-Vector Multiplication”. MA thesis. EECS Department, University of California, Berkeley, Dec. 2014. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-231.html`.

[23] A. Dutt and V. Rokhlin. “Fast Fourier Transforms for Nonequispaced Data”. In: *SIAM Journal on Scientific Computing* 14.6 (1993), pp. 1368–1393. DOI: `10.1137/0914081`. eprint: `https://doi.org/10.1137/0914081`. URL: `https://doi.org/10.1137/0914081`.

[24] J. A. Fessler. *Michigan Image Reconstruction Toolbox*. `https://web.eecs.umich.edu/~fessler/code/`. Accessed: 2018-12-01.

[25] Matteo Frigo and Steven G. Johnson. “FFTW: An adaptive software architecture for the FFT”. In: *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*. Vol. 3. IEEE, 1998, pp. 1381–1384.

[26] Patrick Goodwill et al. “X-Space MPI: Magnetic Nanoparticles for Safe Medical Imaging”. In: *Advanced Materials* 24.28 (2012), pp. 3870–3877. ISSN: 1521-4095. DOI: `10.1002/adma.201200221`. URL: `http://dx.doi.org/10.1002/adma.201200221`.

[27] Michael Grant and Stephen Boyd. *CVX: Matlab Software for Disciplined Convex Programming, version 2.1*. `http://cvxr.com/cvx`. Mar. 2014.

[28] Michael Schacht Hansen and Thomas Sangild Sorensen. “Gadgetron: An open source framework for medical image reconstruction”. In: *Magnetic Resonance in Medicine* 69.6 (2013), pp. 1768–1776. ISSN: 1522-2594. DOI: `10.1002/mrm.24389`. URL: `http://dx.doi.org/10.1002/mrm.24389`.

[29] Felix Heide et al. “ProxImaL: Efficient Image Optimization Using Proximal Algorithms”. In: *ACM Trans. Graph.* 35.4 (July 2016), 84:1–84:15. ISSN: 0730-0301. DOI: `10.1145/2897824.2925875`. URL: `http://doi.acm.org/10.1145/2897824.2925875`.

[30] Hayley Iben et al. “Artistic Simulation of Curly Hair”. In: *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’13. Anaheim, California: ACM, 2013, pp. 63–71. ISBN: 978-1-4503-2132-7. DOI: `10.1145/2485895.2485913`. URL: `http://doi.acm.org/10.1145/2485895.2485913`.

[31] John D. McCalpin. *Memory Bandwidth and System Balance in HPC Systems*. URL: `https://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/`. Nov. 2016.

[32] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 10/12/2017]. 2001–. URL: `http://www.scipy.org/`.

[33] Amir Kamil, Yili Zheng, and Katherine Yelick. “A Local-View Array Library for Partitioned Global Address Space C++ Programs”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY’14. Edinburgh, United Kingdom: ACM, 2014, 26:26–26:31. ISBN: 978-1-4503-2937-8.

[34] Shoaib Ashraf Kamil. “Productive High Performance Parallel Programming with Autotuned Domain-Specific Embedded Languages”. PhD thesis. EECS Department, University of California, Berkeley, Jan. 2013. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-1.html.

[35] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0.

[36] D. Kim et al. “High-performance 3D Compressive Sensing MRI reconstruction”. In: *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*. Aug. 2010, pp. 3321–3324. DOI: 10.1109/IEMBS.2010.5627493.

[37] Fredrik Kjolstad et al. “Simit: A Language for Physical Simulation”. In: *ACM Trans. Graph.* 35.2 (May 2016), 20:1–20:21. ISSN: 0730-0301. DOI: 10.1145/2866569. URL: http://doi.acm.org/10.1145/2866569.

[38] Justin J. Konkle et al. “A Convex Formulation for Magnetic Particle Imaging X-Space Reconstruction”. In: *PLOS ONE* 10.10 (Oct. 2015), pp. 1–15. DOI: 10.1371/journal.pone.0140137. URL: https://doi.org/10.1371/journal.pone.0140137.

[39] Justin J. Konkle et al. “A Convex Formulation for Magnetic Particle Imaging X-Space Reconstruction”. In: *PLOS ONE* 10.10 (Oct. 2015), pp. 1–15. DOI: 10.1371/journal.pone.0140137. URL: https://doi.org/10.1371/journal.pone.0140137.

[40] Hsiou-Yuan Liu, Jingshan Zhong, and Laura Waller. “Multiplexed phase-space imaging for 3D fluorescence microscopy”. In: *Opt. Express* 25.13 (June 2017), pp. 14986–14995. DOI: 10.1364/OE.25.014986. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-25-13-14986.

[41] Michael Lustig and John M. Pauly. “SPIRiT: Iterative self-consistent parallel imaging reconstruction from arbitrary k-space”. In: *Magnetic Resonance in Medicine* 64.2 (2010), pp. 457–471. ISSN: 1522-2594. DOI: 10.1002/mrm.22428. URL: http://dx.doi.org/10.1002/mrm.22428.

[42] Chenguang Ma et al. “Motion deblurring with temporally coded illumination in an LED array microscope”. In: *Opt. Lett.* 40.10 (May 2015), pp. 2281–2284. DOI: 10.1364/OL.40.002281. URL: http://ol.osa.org/abstract.cfm?URI=ol-40-10-2281.

[43] S. Marchesini et al. “SHARP: a distributed, GPU-based ptychographic solver”. *Journal of Applied Crystallography*. 2016.

[44] *MATLAB Optimization Toolbox*. The MathWorks, Natick, MA, USA. 2017.

[45] Jacob Mattingley and Stephen Boyd. “CVXGEN: a code generator for embedded convex optimization”. In: *Optimization and Engineering* 13.1 (Mar. 2012), pp. 1–27. ISSN: 1573-2924. DOI: 10.1007/s11081-011-9176-9. URL: https://doi.org/10.1007/s11081-011-9176-9.

[46] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[47] Lustig Michael, Donoho David, and Pauly John M. "Sparse MRI: The application of compressed sensing for rapid MR imaging". In: *Magnetic Resonance in Medicine* 58.6 (), pp. 1182–1195. DOI: 10.1002/mrm.21391. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.21391. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.21391.

[48] *MIT StarCluster Website*. URL: http://star.mit.edu/cluster/.

[49] Ravi Teja Mullapudi et al. "Automatically Scheduling Halide Image Processing Pipelines". In: *ACM Trans. Graph.* 35.4 (July 2016), 83:1–83:11. ISSN: 0730-0301. DOI: 10.1145/2897824.2925952. URL: http://doi.acm.org/10.1145/2897824.2925952.

[50] Mark Murphy. "Parallelism, Patterns, and Performance in Iterative MRI Reconstruction". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-134.html.

[51] D.G. Nishimura. *Principles of magnetic resonance imaging*. Stanford University, 1996.

[52] R. Orendorff and D. Hensley. *PyOP: Matrix Free Linear Operators*. http://ryan.orendorff.io/pyop/index.html. Accessed: 2018-12-01.

[53] Klaas P. Pruessmann et al. "SENSE: Sensitivity encoding for fast MRI". In: *Magnetic Resonance in Medicine* 42.5 (1999), pp. 952–962. ISSN: 1522-2594. DOI: 10.1002/(SICI)1522-2594(199911)42:5<952::AID-MRM16>3.0.CO;2-S. URL: http://dx.doi.org/10.1002/(SICI)1522-2594(199911)42:5%3C952::AID-MRM16%3E3.0.CO;2-S.

[54] M. Puschel et al. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 232–275. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840306.

[55] Tran Minh Quan et al. "Multi-GPU Reconstruction of Dynamic Compressed Sensing MRI". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III*. Ed. by Nassir Navab et al. Cham: Springer International Publishing, 2015, pp. 484–492. ISBN: 978-3-319-24574-4. DOI: 10.1007/978-3-319-24574-4_58. URL: https://doi.org/10.1007/978-3-319-24574-4_58.

[56] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176. URL: http://doi.acm.org/10.1145/2491956.2462176.

[57] Benjamin Recht et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems 24*. Ed. by J. Shawe-Taylor et al. Curran Associates, Inc., 2011, pp. 693–701. URL: `http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf`.

[58] Hongbo Rong et al. "Sparso: Context-driven Optimizations of Sparse Linear Algebra". In: *Proc. of the 2016 Intl. Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel: ACM, 2016, pp. 247–259. ISBN: 978-1-4503-4121-9. DOI: `10.1145/2967938.2967943`. URL: `http://doi.acm.org/10.1145/2967938.2967943`.

[59] Volker Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. ISSN: 0945-3245. DOI: `10.1007/BF02165411`. URL: `https://doi.org/10.1007/BF02165411`.

[60] Jonathan I. Tamir et al. "T2 shuffling: Sharp, multicontrast, volumetric fast spin-echo imaging". In: *Magnetic Resonance in Medicine* 77.1 (2017), pp. 180–195. DOI: `10.1002/mrm.26102`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.26102`.

[61] Yuan Tang et al. "The Pochoir Stencil Compiler". In: *Proc. of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 117–128. ISBN: 978-1-4503-0743-7. DOI: `10.1145/1989493.1989508`. URL: `http://doi.acm.org/10.1145/1989493.1989508`.

[62] Zhang Tao, Pauly John M., and Levesque Ives R. "Accelerating parameter mapping with a locally low rank constraint". In: *Magnetic Resonance in Medicine* 73.2 (), pp. 655–661. DOI: `10.1002/mrm.25161`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.25161`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.25161`.

[63] *TOP500 Supercomputer Site*. URL: `http://www.top500.org`.

[64] Martin Uecker et al. "Berkeley Advanced Reconstruction Toolbox". In: *Proc. Intl. Soc. Mag. Reson. Med.* Vol. 23. 2015, p. 2486.

[65] Martin Uecker et al. "ESPIRiT—an eigenvalue approach to autocalibrating parallel MRI: Where SENSE meets GRAPPA". In: *Magnetic Resonance in Medicine* 71.3 (2014), pp. 990–1001. ISSN: 1522-2594. DOI: `10.1002/mrm.24751`. URL: `http://dx.doi.org/10.1002/mrm.24751`.

[66] Richard Vuduc, James W Demmel, and Katherine Yelick. "OSKI: A library of automatically tuned sparse matrix kernels". In: 16 (Jan. 2005), pp. 521–530.

[67] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: `10.1145/1498765.1498785`. URL: `http://doi.acm.org/10.1145/1498765.1498785`.

[68] X. L. Wu et al. “Impatient MRI: Illinois Massively Parallel Acceleration Toolkit for image reconstruction with enhanced throughput in MRI”. In: *2011 IEEE Intl. Symposium on Biomedical Imaging*. Mar. 2011, pp. 69–72. DOI: `10.1109/ISBI.2011.5872356`.

[69] Matt Wytock et al. “A New Architecture for Optimization Modeling Frameworks”. In: *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*. PyHPC ’16. Salt Lake City, Utah: IEEE Press, 2016, pp. 36–44. ISBN: 978-1-5090-5220-2. DOI: `10.1109/PyHPC.2016.5`. URL: `https://doi.org/10.1109/PyHPC.2016.5`.

[70] Katherine Yelick et al. “Titanium: a high-performance Java dialect”. In: vol. 10. 11-13, pp. 825–836.

[71] Y. Zheng, M. R. Kristensen, and B. Vinter. “PGAS for Distributed Numerical Python Targeting Multi-core Clusters”. In: *Parallel and Distributed Processing Symposium, International (IPDPS)*. Vol. 00. May 2012, pp. 680–690.