Indigo: A Domain-Specific Language for Fast, Portable Image Reconstruction

Michael Driscoll*†, Benjamin Brock*†, Frank Ong*, Jonathan Tamir*, Hsiou-Yuan Liu*,
Michael Lustig*, Armando Fox*, Katherine Yelick†*
* Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
† Computational Research Division, Lawrence Berkeley National Laboratory
Correspondence: driscoll@cs.berkeley.edu

***Abstract*—Linear operators used in iterative methods like conjugate gradient have typically been implemented either as "matrix-driven" subroutines backed by explicit sparse or dense matrices, or as "matrix-free" subroutines that implement specific linear operations directly (e.g. FFTs). The matrix-driven approach is generally more portable because it can target widely-available BLAS libraries, but it can be inefficient in terms of time and space complexity. In contrast, the matrix-free approach is more performant because it leverages structure in operations, but it requires each operator be re-implemented on each new platform.**

To increase performance and portability, we propose a hybrid approach that represents linear operators as expression trees. Leaf nodes in the tree are either matrix-free or matrix-driven operators, and interior nodes represent mathematical compositions (sums, products, transposes) or structural compositions (stacks, block diagonals, etc.) of the leaf operators. This representation enables expert-guided reordering and fusion transformations that can improve performance or reduce memory pressure. We implement our approach in a domain-specific language called Indigo. We assess Indigo on image reconstruction problems arising in four application areas: magnetic resonance imaging, ptychography, magnetic particle imaging, and fluorescent microscopy. We give performance results from vendor BLAS libraries, and we introduce specializations to Sparse BLAS routines that achieve near-Roofline performance on multi-core, many-core, and GPU systems.

I. Introduction

A number of linear inverse problems arising in computational imaging can be solved via iterative optimization methods. Central to these methods is the definition of a "forward model" linear operator that represents how an intrinsic image is transformed when observed by an imaging system. When the forward operator is used in gradient-based methods such as conjugate gradient, the intrinsic image can be reconstructed from the acquired scan data. Evaluation of the forward operator is often the dominant cost in image reconstruction, so computational performance is critical. At odds with the demand for performance are concerns about extensibility—operators vary across both imaging domains and problem instances from the same domain—and portability—practitioners wish to run on a variety of computational platforms such as multi-core CPUs and GPUs. It remains a challenge to achieve both portability and performance for linear operator implementations.

Operator implementations can be classified into two groups. Early convex optimization frameworks used a "matrix stuffing" approach that expressed linear transformations as explicit sparse or dense matrices. This concrete representation is general enough to express any linear transformation, but it is suboptimal when operators admit fast algorithms. For example, the Cooley-Tukey algorithm can compute an n-length FFT in $O(n)$ space and $O(n \log n)$ operations, versus $O(n^2)$ space and $O(n^2)$ operations for a general matrix-vector product. Later work sought to leverage fast algorithms like FFTs in "matrix-free" implementations that implement linear transformations via fast subroutines. High-performance libraries for common linear transformations are generally available, so it is straightforward to obtain portable performance for single instances of these operators.

To make the challenge more concrete, consider a magnetic resonance imaging (MRI) reconstruction task in which we wish to reconstruct a three-dimensional image representing the density of protons in a volume. Given a forward operator A, a vector of scan data y, and a candidate image vector x, we can use conjugate gradient to solve the system $A^H Ax = A^H y$ for x. Reconstruction time will be dominated by the evaluation of the forward operator A, which can be broken down into two steps. In the first step, the input image is weighted according to a pre-determined spatial sensitivity map that models signal attenuation. In the second step, the weighted image undergoes a centered Fourier Transform implemented as a transpose-like shift operation, a Fast Fourier Transform (FFT), and a second shift. A matrix-driven implementation would compute A as the product of a diagonal matrix S for sensitivity weighting, a sparse matrix M for shifting, and a dense FFT matrix F: $A = MFMS$. In contrast, a matrix-free implementation would represent the operator as the composition of subroutines: $\mathcal{A}(\cdot) = (\mathcal{M} \circ \mathcal{F} \circ \mathcal{M} \circ \mathcal{S})(\cdot)$. Here, the $\mathcal{F}$ routine could be drawn from a standard FFT library, but the $\mathcal{M}$ and $\mathcal{S}$ routines are less standard and would have to be re-implemented on each platform.

Even with fast individual operators, end-to-end performance remains elusive because forward operators typically comprise multiple sub-operators arranged in some manner. Substantial data reuse opportunities exist across operators, so it is insufficient to optimize them in isolation—we must consider operator fusion. With matrix-free operators, automatic fusion might be attainable via static analysis and code generation. However, this approach is subject to the limitations of static analysis and requires substantial machinery to implement. Furthermore, it can be difficult to reason about the performance characteristics of matrix-free operators because of their abstract representation.

We seek to combine the matrix-free and matrix-driven approaches to constructing linear operators. We still leverage

matrix-free subroutines for FFTs and other fast transforms, but we revisit matrix stuffing for "accessory" transforms like padding, cropping, blurring, shifting, and element-wise multiplication. We implement our approach in a language called Indigo. Indigo simplifies analysis and specialization of operators because their structure is statically known, and it aids portability because new platforms must only implement AXPBY, FFT, and matrix multiplication routines. Indigo operators are also amenable to performance modeling because the performance characteristics of the underlying subroutines are well understood. We give a Roofline analysis [34] that reveals Indigo operators run within a significant fraction of machine peak, and we describe techniques for directing future optimization efforts. Indigo is available for download at https://pypi.python.org/pypi/indigo.

This paper is organized as follows. Section II presents the Indigo representation of linear operators. Sections III and IV describe transformations that the compiler and runtime can apply to move between mathematically-equivalent operators with better performance characteristics or smaller memory footprints. Section V explains how profitable transformations are selected from the broad set of possible transformations. Section VI evaluates Indigo on reconstruction tasks from four imaging modalities on three modern shared-memory platforms. Section VII compares Indigo to related work and Section VIII concludes with directions for future work.

II. Domain-Specific Language

Indigo is a domain-specific language for constructing fast, structured linear operators. Its fundamental object is an `Operator` that represents an arbitary linear transformation. `Operators` are broadly categorized as `MatrixOps`, which are backed by explicit matrices, or `MatrixFreeOps`, which use alternate evaluation strategies. `MatrixFreeOPs` can be further classified as `CompositeOps`, which represent arrangements of `Operators`, and `FastRoutineOps`, which call subroutines that implement fast linear transformations. Operator trees can be constructed according to the following grammar.

```
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+ ')' ;
```

We describe in detail the set of operators provided by Indigo.

A. Leaf Operators

To produce an Indigo operator, the application programmer first instantiates operators from the `MatrixOp` and `FastRoutineOp` classes. These operators represent leaves which can be arranged into an operator tree. The most general class of operators is the `MatrixOp`, which represents a matrix implementing any linear transformation. Operators that can be evaluated via matrix-free methods are classified as `FastRoutineOps`. These include `FFTs`, `Identity` matrices, and matrices of ones (`OneMatrix`).

B. Composite Operators

Application programmers next use `CompositeOps` to arrange sub-operators into an expression tree. `CompositeOps` reflect mathematical or structural relationships between operators. Common examples include `Product` and `Sum` operators representing the inner product and sum, respectively, of child matrices; `Adjoint` operators representing the conjugate-transpose of their sub-operator; and `Scale` operators representing their child operator scaled by some scalar value. Structural composite operators include the `BlockDiag` operator, which represents sub-operators arranged along a diagonal; the `VStack` and `HStack` operators, which represent vertically and horizontally stacked sub-operators; and the `KronI` operator, which represents the Kronecker product of an identity matrix and its sub-operator. The `KronI` operator is similar to a `BlockDiag` operator, but captures replication of its sub-operator along the diagonal. Furthermore, evaluation of a `KronI` operator is particularly efficient because the matrix-vector operation

$$vec(Y) = (I_c \otimes A) \cdot vec(X)$$

can be computed as the matrix-matrix operation $Y = AX$ where A, X, and Y are arbitrary matrices, I_c is a identity matrix with edge length c, '$\otimes$' is the Kronecker product, and $vec(\cdot)$ stacks columns of a matrix into a vector.

C. Derived Operators

The previous two subsections presented the operator classes implemented by Indigo. These classes are general enough to derive a variety of other operators. Derived operators are instantiated via factory functions provided by Indigo.

1) Matrix-Stuffed Derived Operators: When no operator exists for a particular operation, one can be created via matrix-stuffing. Indigo provides factory routines for building sparse matrices that perform common operations like element-wise multiplication, padding, cropping, and blurring. The factory routines can be parameterized to enhance generality. For example, the factory routine for building a padding operator accepts both a pad width and a padding scheme (center-padded or end-padded). These factory routines all return `MatrixOps`.

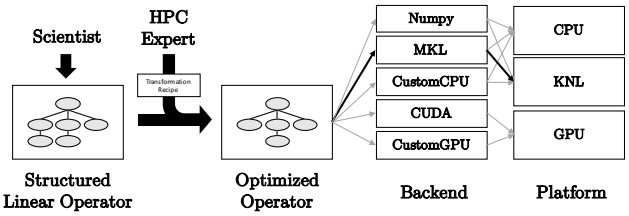


Fig. 1: To use Indigo, an application programmer constructs a linear operator using `Operator` classes provided by Indigo. The operator is then optimized by a transformation recipe, written by a performance expert, into a mathematically equivalent operator tree with better performance characteristics. Finally, the tree is evaluated by an Indigo backend on one of several computational platforms.

2) Composite Derived Operators: Additional factory routines implement higher-level functionality by constructing full-fledged operator trees, rather than single operator instances. For example, Indigo implements a unitary Fourier Transform as `FFT` operator modified by a `Scale` operator whose scale factor is informed by the FFT dimensions. Likewise, a centered FFT is implemented by left- and right-multiplying an `FFT` with diagonal matrices that perform FFT-shifts. A non-uniform Fourier Transform [9] is implemented as a product of an "apodization" matrix, a centered, unitary `FFT`, and an interpolation matrix.

D. Workflow

To use Indigo, an application programmer constructs an operator tree by instantiating operator classes. Next, the operator tree is specialized according to a transformation recipe written by a performance expert. The optimized operator is then available for use in the mathematical optimization routine of the user's choice. Indigo provides implementations of the conjugate gradient and FISTA methods [4], though more are possible. Typically, the mathematical optimization routines alternate between linear operator evaluations and BLAS-1 evaluations specific to the particular routine. On each evaluation, the operator tree is interpreted by an Indigo backend on CPU, KNL, or GPU platforms. This process is illustrated in Figure 1.

III. Operator Specialization

The Indigo representation of linear operators enables transformations between numerically-equivalent operators that possess different computational performance and memory footprint characteristics. Indigo employs three flavors of operator specialization: tree transformations, operator realization, and matrix inspection.

A. Tree Transformations

Indigo operator trees can be reorganized according to common algebraic principles. These transformations have minimal immediate effects on performance; however, they can enable profitable transformations later in the specialization process.

Many `CompositeOps` admit a distributive property. There are many such rules—we count up to 504—so we sketch a few here. The remaining ones can be reasoned about straightforwardly. For example, the `KronI` operator distributes over a `Product`:

$$I_c \otimes (A \cdot B) = (I_c \otimes A) \cdot (I_c \otimes B).$$

Some products of structural operators can be reorganized if the dimensions of the child operators match. Examples include the matrix-vector-like operation

$$BlockDiag(A, B) \cdot VStack(C, D) = BlockDiag(AC, BD)$$

or the inner-product-like

$$HStack(A, B) \cdot VStack(C, D) = [AB + CD]\,.$$

The `Sum` and `Product` operators admit an associative property, which allows tree rotations:

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

`HStack` and `VStack` operators are related by adjoints:

$$HStack(A, B) = VStack(A^H, B^H)^H.$$

Finally, the `Adjoint` operator admits an inverse:

$$Adjoint(Adjoint(A)) = A.$$

It is worth noting that all of these transforms are bi-directional.

B. Realization Transformations

The second major category of specializations is operator realization, which converts matrix-free operators into explicit matrices. Many realization transformations seem inadvisable because they increase the memory footprint and/or inhibit performance. However, like tree transformations described previously, they can enable other, more profitable transformations. Each `MatrixFreeOp` has an associated realization transformation, described here. In order to be eligible for realization, a `CompositeOp's` children must all be realized matrices (`MatrixOps`).

`Product` operators are realized via matrix-matrix multiplication. The performance implications of this transformation depend on the sparsity structure of the underlying matrices, since the product of two sparse matrices could be more or less sparse than the original matrices. We've found in practice that realizing `Products` tends to maintain sparsity because the sparsity pattern reflects some real-world structure. One example that illustrates this effect can be found in the NUFFT, which contains a product of an FFT-shift matrix and an interpolation matrix—the FFT-shift matrix is diagonal, so multiplying it by the interpolation matrix yields a new matrix with identical structure to the original interpolation matrix. We liken this transformation to operator fusion.

The `HStack`, `VStack`, and `BlockDiag` operators are realized by assembling a new, larger matrix from the constituent blocks. Performance and memory footprint are largely unchanged.

The `KronI` operator is realized by explicitly replicating its child matrix. The realized matrix is larger by the replication

factor, and immediate performance likely suffers by the same factor.

The `Adjoint` operator is realized by explicitly transposing and conjugating the underlying matrix. This transformation can have a substantial effect on performance since Indigo primarily uses a compressed sparse row (CSR) storage format for sparse matrices. For certain sparsity structures, a CSR format maintains better temporal locality in the input or output vectors.

The `Scale` operator is realized by multiplying all elements in its child matrix by its scale factor.

It is theoretically possible to realize `FFT` operators, but typically impractical to do so on realistic problems because the resulting matrix would be large and dense. The MRI application described in Section VI would require 41 petabytes to store its 3D-FFT matrix.

C. Matrix Inspection

The third major category of specialization is an inspection step in which Indigo examines the remaining matrices in search of structural properties that enable faster evaluation than a general SpMM routine can achieve. In particular, Indigo checks matrices for properties we term *row* or *column write exclusivity*. Row write exclusivity holds when a matrix has no more than one nonzero per row (or per column for column write exclusivity). The presence of write exclusivity allows matrix multiplication routines to avoid synchronization in some operations. More specifically, a sparse matrix multiplication routine (SpMM) computing the product of an implicitly-transposed, column-exclusive, CSR-format matrix needn't synchronize on accumulations into the output vector because there won't be more than one element accumulated. This optimization is especially important on GPU platforms where atomics are relatively costly.

The row and column write exclusivity properties are cheap to discover. Indigo determines if a CSR matrix is row exclusive by a single traversal of the *row_ptr* vector (in Netlib terminology [1]); a matrix is row exclusive if no adjacent row pointers differ by more than one element. Determining column exclusivity is slightly more difficult. Indigo computes a histogram of column indices (the *col_ind* array) and certifies exclusivity if no histogram bin has more than one element. This procedure is similar in cost to one matrix-vector product. Indigo determines exclusivity information once, as the final specialization step.

IV. Runtime Optimizations

The Indigo runtime evaluates an operator tree much like an interpreter traverses an abstract syntax tree. During evaluation, a *batch size* parameter enables a trade-off between performance and memory footprint. To understand the effects of the batch size, consider a scenario in which an operator is right-multiplied by a dense matrix with N columns. The runtime can schedule this as one sparse-matrix-times-dense-matrix (SpMM) on N columns, N SpMVs on 1 column, or generally $\frac{N}{b}$ SpMMs on b columns. These cases can have very different memory requirements depending on the structure of the Indigo operator. If the operator tree contains a `Product` node, then memory proportional to b must be allocated to hold the intermediate result. Similarly, if the tree contains an FFT, the underlying FFT library might allocate scratch space proportional to b. In memory-limited settings, it is desirable to minimize these allocations so the problem instance fits in memory. To this end, the runtime can use a smaller batch size b to maintain residency, at the cost of some performance. Batch sizes can be set at any node in the operator tree.

V. Transformation Recipes

So far, we have described the space of transformations that can be applied to an operator tree to change its performance characteristics. It remains an open question how to choose good transformations that yield fast operators, operators with small memory footprints, or some balance thereof. Automatic methods guided by heuristics or empirical tuning might succeed in finding good configurations, but several factors compound the search problem. First, the optimization space is highly non-smooth. A good tree is often found by applying a long series of seemingly pointless or suboptimal transformations. Second, it's difficult to enumerate the search space because the tree transformations are bi-directional, so care must be taken to identify equivalent, previously-visited points. Third, many of the transformations cannot be assessed without performing them. In particular, it's difficult to predict whether multiplying two matrices will yield a new matrix that's faster or smaller than its predecessors. One could compute the new matrix via matrix-matrix multiplication, but it significantly slows down a search procedure.

Indigo instead allows a performance expert to devise a transformation recipe that is applied to operator trees constructed by an application programmer. A transformation recipe is a series of imperative statements that perform tree manipulations such as those described in Section III. Indigo implements a number of transformations, and new ones can be written by the user. We have found that transformation recipes are general enough to cover variation among trees from a particular application area, so it is not necessary to rewrite them frequently. They also enable experts to quickly try out new configurations without substantial programming effort. The use of transformation recipes does not rule out automatic techniques, as future transformation recipes could be generated by an autotuner or heuristic engine.

To illustrate a transformation recipe, we return to our running MRI example. Recall the operator $A := (MFM)S$, which represents the product of a centered FFT and a diagonal matrix of sensitivity weights. One transformation recipe likely to yield good performance is:

```
AssociateSpMats() // yields (M)F(MS)
RealizeProds() // yields (M)F(Z), Z = MS
```

The first transformation performs tree rotations to group sparse matrices. In particular, it rotates the second centering matrix M away from the matrix-free FFT operator F, and toward

ID	Platform	Peak Flops	Memory Bandwidth
CPU	Intel Xeon E5-2698 v3	1.4 TFlop/s	144 GB/s
KNL	Intel Xeon Phi 7250	3 TFlop/s	464 GB/s
GPU	NVIDIA Titan X	10 TFlop/s	360 GB/s

TABLE I: Indigo evaluation platform characteristics. We measure memory bandwidth via a call to each vendor's AXPBY routine. On KNL, our working set is small enough to fit in MCDRAM, so we cite MCDRAM bandwidth rather than main memory bandwidth.

the sensitivity weighting matrix S. The second step multiplies S and M, effectively fusing the sensitivity weighting and centering operations. Since S is diagonal, $M \cdot S$ is identical in structure to M. The resulting operator ($A' = MFZ$) has better performance and a smaller memory footprint than the original A.

VI. Performance Evaluation

We evaluate Indigo on reconstruction problems from four imaging modalities: magnetic resonance imaging (MRI), ptychography, fluorescent microscopy, and magnetic particle imaging (MPI). Indigo implements five backends that we test on the CPU, GPU, and KNL platforms described in detail in Table I. Our backends include:

- **numpy** : A reference backend implemented on Numpy. It is portable and compatible with Python debugging tools.
- **mkl** : A backend implemented on Intel MKL. It runs on the CPU and KNL platforms.
- **cuda** : A backend implemented on NVIDIA's cuFFT and cuSPARSE libraries. It runs on the GPU platform.
- **customcpu** : A high-performance backend that implements exclusivity-aware SpMM routines in C and OpenMP. It uses MKL for its FFTs, and it runs on CPU and KNL platforms.
- **customgpu** : A high-performance GPU backend that implements exclusivity-aware SpMM routines in CUDA. It uses cuFFT for its FFTs.

We seek to characterize the performance of Indigo as a fraction of the peak performance defined by the Roofline model [34]. This statistic captures how well our implementation fulfills the goals of our implementation strategy (i.e. mixing matrix and matrix-free operators). For the MRI application, we also assess how well our implementation strategy performs relative to an optimized matrix-free implementation.

To determine the fraction of Roofline peak for our codes, we compute the arithmetic intensity of our kernels as follows. For three-dimensional FFTs of shape N^3, we expect $5N^3 \log(N^3)$ flops (per the FFTW model [10]) and $6N^3$ elements of memory traffic (one read and write of the dataset is required per dimension when no N^2 slab fits in cache). For all other operations (OneMMs, AXPBYs, CSRMMs), we expect them to run at the STREAM [23] bandwidth: they should read the input data once and write the data once. We also consider sparse matrix structure—we account for input elements that need not be read, and output elements that need not be written.

A. Magnetic Resonance Imaging

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 [19], [30]. 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.

1) Background: 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-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 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 [25] 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 number of sensitivity maps S of the same dimensions, yielding an array of channel-images k with shape (c, x, y, z). Then, each channel-image undergoes a non-uniform Fourier transform $\mathscr{F}$ to yield the acquired data k. Mathematically, the forward SENSE operation is

$$k_i = \mathscr{F} S_i \cdot \rho, \quad \forall i \in c \tag{1}$$

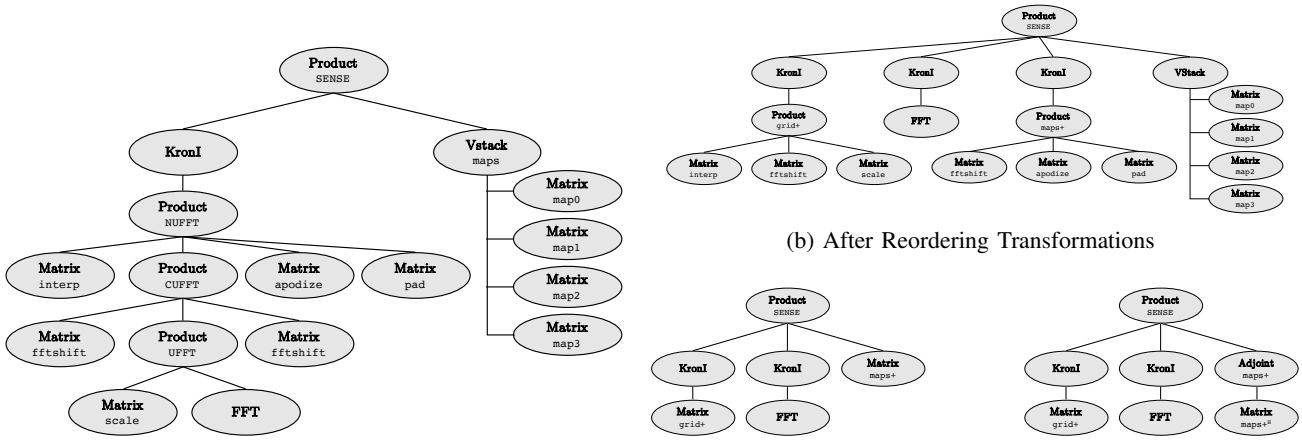
and the adjoint operation is

$$\rho = \sum_{i=1}^{c} S_i^H \cdot \mathscr{F}^H \cdot k_i. \tag{2}$$

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 corresponding Indigo operator tree is illustrated in Figure 2a. The adjoint operator is implicitly defined at the same time as the conjugate transpose of the forward operator. When used with ℓ-1 regularization techniques, the SENSE operator is sufficient for implementing a reconstruction pipeline for compressed sensing MRI.



(a) User-Provided Operator Tree

(b) After Reordering Transformations

(c) After Operator Realization

(d) After `maps+` Transpose

Fig. 2: Transformation of the SENSE operator tree. The user-provided tree (a) is first reordered to lift matrix-free FFT nodes by distributing the `KronI` 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 SpMMs and one batched FFT.

2) Challenges: The user-defined operator tree (Figure 2a) 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 would like 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 2b. 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 2c.

A performance analysis of the tree in Figure 2c 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 2d. 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 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 SpMM multiplication routines that perform non-atomic accumulations, rather than atomic ones.

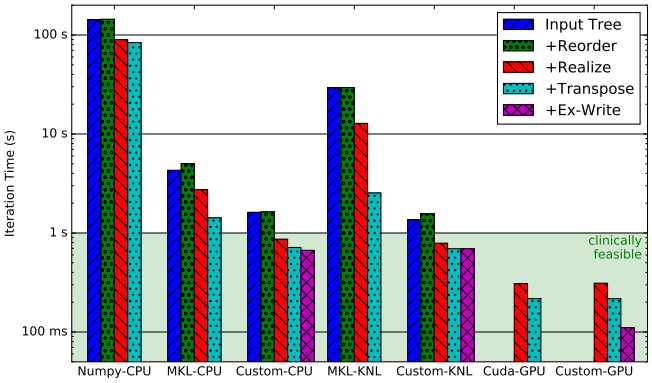


Fig. 3: The performance of each intermediate SENSE operator during the specialization process. The operator tree undergoes four major transformations: reordering, realization, a transpose of the `maps` matrix, and discovery of the exclusive write property for the transposed `maps` matrix. Realization is required on the GPU to fit into the 12GB of available memory, hence we omit the first two results. We also omit the write exclusivity result for backends which don't leverage it. We denote the region in which operators are fast enough to be clinically viable with a green background.

This results in a $10\times$ speedup on the adjoint evaluation of the `maps+` matrix, and a $2\times$ speedup overall.

3) Performance: The Indigo SENSE operator achieves 91% of the Roofline peak on the GPU platform, and the reconstruction task finishes within one minute. Figure 3 illustrates the performance of the SENSE operator during the transformation process, and Table II gives the performance breakdown of the fastest SENSE operator 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

Platform		FFTs			SpMMs			Overall		
		GFlops/s	%Roofline	%Time	GFlops/s	%Roofline	%Time	GFlops/s	%Roofline	Time
CPU	Numpy	1	0%	86%	0	1%	14%	1	0%	87,040 ms
	MKL	162	41%	31%	1	9%	69%	51	19%	1,321 ms
	Custom	157	40%	67%	4	37%	33%	107	39%	743 ms
KNL	MKL	213	11%	13%	0	1%	87%	27	2%	1,573 ms
	Custom	207	11%	59%	4	8%	41%	124	10%	536 ms
GPU	CUDA	934	95%	34%	8	16%	66%	327	44%	205 ms
	Custom	932	95%	72%	37	80%	28%	680	91%	114 ms

TABLE II: Performance breakdown of one iteration of the SENSE normal operator for MR image reconstruction (Section VI-A). 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.

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.

4) Comparison to Matrix-Free SENSE: We can characterize the performance of a fully-matrix-free SENSE implementation (rather than our hybrid approach) 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, the standard representation in the field. 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.

B. 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 [11]. In a simplified sense, pixel data are acquired in a series of overlapping panels that represent subsets of the intrinsic image, but panels have their average values removed by 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 [17]. They propose a forward operator $A = DS$, where S selects overlapping panels of the image and D computes and subtracts the average value of each panel. Visually, the operators are

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

$\mathbb{1}$ 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 `MatrixFreeOps`. The MPI operator is different from our other applications because it doesn't perform any FFTs. Instead, the forward operator reduces to an AXPBY, an SpMM (with the S matrix), and a OneMM. We evaluate the operator on a 1024-by-768 pixel image divided into 12 panels. Performance results are given in Table III. 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.

C. Ptychography

Ptychography is an imaging modality which uses patterns from diffracted light to reconstruct an image of an object [20]. 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

Platform		AXBPYs %Peak	AXBPYs %Time	SpMMs %Peak	SpMMs %Time	OneMMs %Peak	OneMMs %Time	Overall %Peak	Overall 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 III: Performance breakdown of one iteration of the Magnetic Particle Imaging (MPI) reconstruction operator (Section VI-B). 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.

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 IV: Performance breakdown of one iteration of the Ptychography reconstruction operator (Section VI-C). Fraction of peak data are with respect to the Roofline peak for each respective operation.

X-ray beam at the object from different angles. The diffraction of the X-rays can be described by the equation

$$\mathbf{a} = |\mathbf{F}\mathbf{Q}\psi^{\vee}|$$

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

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

Note that this includes the nonlinear term $|\mathbf{F}\mathbf{z}_i|$. 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 IV shows the performance of this solution. Since matrix structure does not benefit from 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.

D. Phase-Space 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 [18], 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.

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

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 V 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.

VII. Related Work

Indigo builds on ideas from domain-specific languages, sparse linear algebra, and high-performance medical imaging.

a) 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 [27], Simit for finite-element simulations [16], Pochoir for stencils [29], Spiral for

Platform		FFTs %Peak	SpMMs %Peak	Overall %FFTs	Overall %Peak	Overall 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 V: Performance breakdown of one iteration of the phase-space microscopy reconstruction operator (Section VI-D). 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.

digital signal processing [26], and Opt for image optimization problems [8]. These languages share the technique of restricting generality to aid analysis and transformation.

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 [12]. CVXGEN generates C code that implements a custom quadratic program solver designed for use in embedded devices [22]. Chu et al. also explore code generation for convex problems [7], but focus primarily on correctness than performance. Cvxflow [36] and ProxImaL [14] 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-node optimizations within the subset of nodes that represent the linear operator.

Our technique of separating the transformation recipe from the code to be transformed has been explored in other DSLs. We examine two: CHiLL [2] and Halide [27]. 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 [24] and affirms the transformation recipe approach.

b) Sparse Linear Algebra: Indigo's collection of linear operators is similar to those provided by the SciPy [15] and Matlab [21]. 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 [32] uses empirical performance tuning to select block sizes on a per-platform basis. Sparso [28] exploits optimization opportunities across sparse matrix operations, but its implementation in MKL yielded no benefit on our matrices. The CUDA sparse BLAS library [5] 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.

c) High Performance Image Reconstruction: Previous work has sought to reduce time-to-solution in a variety of iterative reconstruction problems. One rewarding approach has been algorithmic modifications that offer different performance profiles [33], or that trade accuracy for performance [3]. Another body of work has focused on fast, ad-hoc implementations of reconstruction codes; these include Bart [31], PowerGrid [6], Impatient [35], and Gadgetron [13] for MRI, and SHARP for ptychography [20].

VIII. Discussion and Conclusion

The paper introduces Indigo, an embedded domain-specific language for implementing fast image reconstruction codes. By restricting users to a limited set of concepts (linear operations), Indigo separates concerns about algorithmic specification and performance optimization and enables portable performance. We demonstrate Indigo on four applications—magnetic resonance imaging, magnetic particle imaging, fluorescent microscopy, and ptychography—and show that Indigo achieves a substantial fraction of peak across a variety of backends without changes to the source code.

The Indigo approach is not without limitations. First, it avoids the challenge of automatic optimization by relying on transformation recipes written by performance experts. To be widely useful, it might be necessary to have generic recipes that yield good average performance, or an analysis engine that can synthesize good recipes. Secondly, real-world reconstructions can employ an expensive nonlinear regularization like total variation or locally-low-rank metrics. These fall outside the domain of Indigo, but they face similar portability and performance challenges. We wrote our regularization routines in OpenMP-parallelized C code. Third, construction of Indigo operators can be unintuitive because it entails programmatic construction of a abstract syntax trees rather than construction via a language syntax and parser. Fortunately, mathematical and visual structure abounds in this domain, and we are optimistic that better interfaces can facilitate faster and more correct operator construction.

One final benefit of Indigo is its ability to reduce a collection of complex, application-specific linear operators to a handful of low-level mathematical operations (SpMMs, OneMMs, AXPBYs, and FFTs). This simplifies the task of performance optimization, which can be informed by extensive work on fast linear algebra, and enables optimizations developed for one application can propagate to other applications without source-level changes. The Indigo representation also allows performance experts to search for well-performing implementations faster than if they were developing them from scratch. Ultimately, Indigo aims to advance the state of science and medicine by making image reconstruction codes faster and more widely available.

ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research was partially funded by DARPA Award Number HR0011-12-2-0016, by NIH grants R01-EB009690 and U01-EB025162, by the Bakar family fund and by ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[2] Protonu Basu, Samuel Williams, Anand Venkat, Brian Straalen, Mary Hall, and Leonid Oliker. Compiler generation and autotuning of communication-avoiding operators for geometric multigrid, 12 2013.

[3] P. J. Beatty, D. G. Nishimura, and J. M. Pauly. Rapid gridding reconstruction with a minimal oversampling ratio. *IEEE Transactions on Medical Imaging*, 24(6):799–808, June 2005.

[4] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Img. Sci.*, 2(1):183–202, March 2009.

[5] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[6] A. Cerjanic, J. L. Holtrop, G-C. Ngo, B. Leback, G. Arnold, M. Van Moer, G. LaBelle, J. A. Fessler, and B. P. Sutton. PowerGrid: A open source library for accelerated iterative magnetic resonance image reconstruction. In *Proc. Intl. Soc. Mag. Res. Med.*, page 525, 2016.

[7] Eric Chu, Neal Parikh, Alexander Domahidi, and Stephen Boyd. Code generation for embedded second-order cone programming. 2013.

[8] Zachary DeVito, Michael Mara, Michael Zollöfer, Gilbert Bernstein, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *arXiv:1604.06525*, 2016.

[9] A. Dutt and V. Rokhlin. Fast fourier transforms for nonequispaced data. *SIAM Journal on Scientific Computing*, 14(6):1368–1393, 1993.

[10] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] Patrick Goodwill, Emine Saritas, Laura Croft, Tyson N. Kim, Kannan M. Krishnan, David V. Schaffer, and Steven M. Conolly. X-space mpi: Magnetic nanoparticles for safe medical imaging. *Advanced Materials*, 24(28):3870–3877, 2012.

[12] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. http://cvxr.com/cvx, March 2014.

[13] Michael Schacht Hansen and Thomas Sangild Srensen. Gadgetron: An open source framework for medical image reconstruction. *Magnetic Resonance in Medicine*, 69(6):1768–1776, 2013.

[14] Felix Heide, Steven Diamond, Matthias Niessner, Jonathan Ragan-Kelley, Wolfgang Heidrich, and Gordon Wetzstein. Proximal: Efficient image optimization using proximal algorithms. *ACM Trans. Graph.*, 35(4):84:1–84:15, July 2016.

[15] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 10/12/2017].

[16] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2):20:1–20:21, May 2016.

[17] Justin J. Konkle, Patrick W. Goodwill, Daniel W. Hensley, Ryan D. Orendorff, Michael Lustig, and Steven M. Conolly. A convex formulation for magnetic particle imaging x-space reconstruction. *PLOS ONE*, 10(10):1–15, 10 2015.

[18] Hsiou-Yuan Liu, Jingshan Zhong, and Laura Waller. Multiplexed phase-space imaging for 3d fluorescence microscopy. *Opt. Express*, 25(13):14986–14995, Jun 2017.

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

[20] S. Marchesini, H. Krishnan, D. A. Shapiro, T. Perciano, J. A. Sethian, B. J. Daurer, and F. R. N. C. Maia. SHARP: a distributed, GPU-based ptychographic solver. *Journal of Applied Crystallography*, 2016.

[21] Matlab optimization toolbox, 2017.

[22] Jacob Mattingley and Stephen Boyd. Cvxgen: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, Mar 2012.

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

[24] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[25] Klaas P. Pruessmann, Markus Weiger, Markus B. Scheidegger, and Peter Boesiger. Sense: Sensitivity encoding for fast mri. *Magnetic Resonance in Medicine*, 42(5):952–962, 1999.

[26] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, Feb 2005.

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 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, pages 519–530, New York, NY, USA, 2013. ACM.

[28] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. Sparso: Context-driven optimizations of sparse linear algebra. In *Proc. of the 2016 Intl. Conference on Parallel Architectures and Compilation*, PACT '16, pages 247–259, New York, NY, USA, 2016. ACM.

[29] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proc. of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY., 2011. ACM.

[30] Martin Uecker, Peng Lai, Mark J. Murphy, Patrick Virtue, Michael Elad, John M. Pauly, Shreyas S. Vasanawala, and Michael Lustig. Espiritan eigenvalue approach to autocalibrating parallel mri: Where sense meets grappa. *Magnetic Resonance in Medicine*, 71(3):990–1001, 2014.

[31] Martin Uecker, Frank Ong, Jonathan I Tamir, Dara Bahri, Patrick Virtue, Joseph Y Cheng, Tao Zhang, and Michael Lustig. Berkeley advanced reconstruction toolbox. In *Proc. Intl. Soc. Mag. Reson. Med*, volume 23, page 2486, 2015.

[32] Richard Vuduc, James W Demmel, and Katherine Yelick. Oski: A library of automatically tuned sparse matrix kernels. 16:521–530, 01 2005.

[33] FTAW Wajer and KP Pruessmann. Major speedup of reconstruction for sensitivity encoding with arbitrary trajectories. In *Proc. Intl. Soc. Mag. Res. Med*, page 767, 2001.

[34] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[35] X. L. Wu, J. Gai, F. Lam, M. Fu, J. P. Haldar, Y. Zhuo, Z. P. Liang, W. M. Hwu, and B. P. Sutton. Impatient mri: Illinois massively parallel acceleration toolkit for image reconstruction with enhanced throughput in mri. In *2011 IEEE Intl. Symposium on Biomedical Imaging*, pages 69–72, March 2011.

[36] Matt Wytock, Steven Diamond, Felix Heide, and Stephen Boyd. A new architecture for optimization modeling frameworks. In *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '16, pages 36–44, Piscataway, NJ, USA, 2016. IEEE Press.