An interface for a self-optimizing sparse matrix kernel library

Richard Vuduc and James W. Demmel and Katherine A. Yelick
{richie,demmel,yelick}@cs.berkeley.edu

March 14, 2005

Abstract

The BeBOP Optimized Sparse Kernel Interface (OSKI) is a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices, for use by solver libraries and applications. These kernels include sparse matrix-vector multiply and sparse triangular solve, among others. The primary aim of this interface is to hide the complex decision-making process needed to tune a kernel implementation for a particular user's sparse matrix and machine, while also exposing the steps and potentially non-trivial costs of tuning at run-time. Our interface also allows for optional continuous profiling and periodic re-tuning, as well as user inspection and control of the tuning process. This document presents and justifies the specific details of this interface based on our experience in implementing automatically tuned sparse kernels on modern cache-based superscalar machines.

Contents

List of symbols

BCSR	Block compressed sparse row format
VBR	Variable block row format
UBCSR	Unaligned block compressed sparse row format
BLAS	Basic Linear Algebra Subroutines
CSC	Compressed sparse column format
CSR	Compressed sparse row format
FEM	Finite element method
SpMV	Sparse matrix-vector multiply
SpMM	Sparse matrix-multiple vector multiply
SpTS	Sparse triangular solve
SpTSM	Sparse triangular solve with multiple right-hand sides
BiCG	Biconjugate gradient

List of Tables

Listings

1 Goals and Motivation

We present and justify the BeBOP Optimized Sparse Kernel Interface (OSKI), a collection of low-level primitives that provide solver libraries and applications with *automatically tuned* computational kernels on sparse matrices. These kernels include sparse matrix-vector multiply (SpMV) and sparse triangular solve (SpTS), among others. Tuning refers to the process of selecting the data structure and code transformations that lead to the fastest implementation, given a kernel, machine, and matrix. The challenge is that we must often defer tuning until run-time, since the matrix may be unknown until then. This need for run-time tuning differs significantly from the case of dense kernels, where only install- or compile-time tuning has proved sufficient in practice [11, 50].

Our interface reflects the need for and cost of run-time tuning, as extensively documented in our recent work on automatic tuning of sparse kernels [28, 46, 49, 30, 35, 47, 48, 25, 1]. We summarize 6 goals of our interface and the key findings behind each as follows:

1. **Provide basic sparse kernel "building blocks"**: We define an interface for basic sparse operations like SpMV and SpTS, in the spirit of the widely-used Basic Linear Algebra Subroutines (BLAS) [12]. We choose the performance-critical kernels needed by sparse solver libraries and applications (particularly those based on iterative solution methods). The target "users" are sparse solver library writers, and any other user interested in performance-aware programming at the level of the BLAS.

   The recent Sparse BLAS Standard [16, 12] inspired the OSKI design. The main differences are (i) we do not specify primitives for matrix construction, and instead assume that the user can provide an assembled matrix in one of a few standard formats, and (ii) we include explicit support for tuning, as discussed below. The current OSKI interface includes ideas from an earlier design [49, Chapter 8].

2. **Hide the complex process of tuning**: Matrices in our interface are represented by handles, thereby enabling the library to choose the data structure. This indirection is needed because the best data structure and code transformations on modern hardware may be difficult to determine, even in seemingly simple cases [28, 49].

   For instance, many sparse matrices from applications have a natural block structure that can be exploited by storing the matrix as a collection of blocks. For SpMV, doing so enhances spatial and temporal locality. However, we have observed cases in which SpMV on a matrix with an "obvious" block structure nevertheless runs in 38% of the time of a conventional implementation ($2.6\times$ speedup) using a different, non-obvious block structure [49]. Furthermore, we have shown that if a matrix has no obvious block structure, SpMV can still execute in half the time ($2\times$ speedup) of a conventional implementation by *imposing* block structure through explicitly stored zeros, even though doing so results in extra work (flops) [49].

3. **Offer higher-level memory hierarchy-friendly kernels**: The particular kernels defined in our interface are a superset of those available in similar library interfaces, including the recent Sparse BLAS standard [16, 17] and the SPARSKIT library [39], among others [19]. These additional "higher-level" kernels have inherently more opportunities for reuse and can execute much faster than their equivalent implementations in terms of the "standard" kernels.

   For example, in addition to the SpMV operation $y \leftarrow A \cdot x$, we include the kernel $y \leftarrow A^T A \cdot x$ in which A may be read from main memory only once. Compared to a register-blocked two-step implementation, $t \leftarrow A \cdot x, y \leftarrow A^T \cdot t$, a cache-interleaved implementation can be up to $1.8\times$ faster, and up to $4.2\times$ faster than an unblocked two-step implementation [47].

4. **Expose the cost of tuning**: We require the user to request tuning explicitly; in the case of SpMV, tuning can cost $40\times$ as much as a single SpMV operation [49] and so should only be done when the user expects sufficiently many calls to SpMV to justify the cost [49].[1]

   The explicit tuning call is optional if the user does not desire tuning, but may also be called repeatedly to re-tune periodically (see Goal 5 below).

5. **Support self-profiling**: The user cannot always *a priori* predict, say, the number of SpMV operations that will occur during an application run. We designed our interface to allow the library to monitor transparently all operations performed on a given matrix and use this information in deciding how aggressively (*i.e.*, how much) to tune. In principle, self-profiling enables the library to guess whether tuning will be profitable (see Section 2.3 on page 12).

6. **Allow for user inspection and control of the tuning process**: To help the user reduce the cost of tuning, the interface provides two mechanisms allowing her both to guide and to see the results of the tuning process.

   First, the user may provide explicit hints about the workload (*e.g.*, the number of SpMVs) and the kind of structure she believes the matrix possesses (*e.g.*, uniform blocks of size 3×3, diagonals).

   Second, the user may retrieve string-based summaries of what tuning transformations and other performance optimizations have been applied to a given matrix. Thus, a user may see and save these results for re-application on future problems (matrices) which the user believes has similar structure to a previously tuned matrix (Section 3.5 on page 24). Moreover, Section 5 on page 29 discusses a format for these summaries which allows a user to select optimizations manually, thereby allowing her to experiment with alternative or experimental optimizations.

We use a library because it enables the use of run-time information for tuning, and because of its potential immediate impact on applications. Our choice of primitives should integrate readily into popular sparse solver libraries such as PETSc [6, 5], or even high-level problem solving environments such as MATLAB [43, 22].[2] We briefly discuss integration with some systems such as PETSc in Appendix C on page 66. The library-based approach complements other sophisticated techniques for generating sparse kernels, as discussed in Section 6 on page 32.

2 An Introduction to the Tuning Interface by Example

This section introduces the C version[3] of the OSKI interface by a series of examples. The interface uses an object-oriented calling style, where the two main object types are (1) a sparse matrix object, and (2) a dense (multiple) vector object. We anticipate that users will

[1]The cost of tuning is acceptable in important application contexts like solving linear systems or computing eigenvalues by iterative methods, where hundreds (or more) of SpMVs can be required for a given matrix [7, 15, 4, 40].

[2]Indeed, the ATLAS self-tuning library for the dense BLAS [50] and the FFTW library for discrete Fourier transforms [20] are included in MATLAB.

[3]Fortran interfaces are also available.

use the library in different ways, so this section illustrates the library's major design points by discussing three such ways (Sections 2.1–2.3):

1. **Gradual migration to tunable matrix objects and kernels** (Section 2.1): To ease the software development effort when switching an application to use OSKI, the interface supports matrix data sharing when the user's sparse matrix starts in a standard array implementation of some basic sparse matrix format, *e.g.*, compressed sparse row (CSR) format and compressed sparse column (CSC) format. Furthermore, users do not have to use any of the automatic tuning facilities, or may introduce the use of tuned operations gradually over time.

2. **Tuning using explicit workload and structural hints** (Section 2.2 on page 10): Any information the user can provide *a priori* is information the library in principle does not need to rediscover, thereby reducing the overhead of tuning. In this case, users may provide the library with structured hints to describe, for example, the expected workload (*i.e.*, which kernels will be used and how frequently), or whether there is special non-zero structure (*e.g.*, uniformly aligned dense blocks, symmetry). The user calls a special "tune routine" to choose a new data structure performance-optimized for the specified workload.

3. **Tuning using an implicit workload** (Section 2.3 on page 12): The library needs to know the anticipated workload to decide when the overhead of tuning can be amortized, but the user cannot always estimate this workload before execution. Rather than specifying the workload explicitly, a user may rely on the library to monitor kernel calls to determine the workload dynamically. The user must still explicitly call the tune routine to perform optimizations, but this routine optimizes based on an inferred workload.

In either 2 or 3, a user may re-tune periodically by repeatedly calling the tune routine.

Section 3 on page 13 summarizes the interface bindings, and the complete C bindings appear in Appendix B on page 40. We discuss error-handling mechanisms in detail in Section 3.6 on page 24.

2.1 Basic usage: gradually migrating applications

Listing 1 on the next page presents a simple example in C of computing one SpMV *without* any tuning using our interface. This example shows how a user may gradually migrate her code to use our interface, provided the application uses "standard" representations of sparse matrices and dense vectors.

The sparse matrix in Listing 1 on the following page is a 3×3 lower triangular matrix with all ones on the diagonal. The input matrix, here declared statically in lines 6–8, is stored in a CSR format using 2 integer arrays, Aptr and Aind, to represent the non-zero pattern and one array of doubles, Aval, to store the non-zero values. The diagonal is not stored explicitly. This representation is a "standard" way of implementing CSR format in various sparse libraries [39, 38, 5]. This particular example assumes the convention of 0-based indices and does not store the diagonal explicitly.

Lines 9–10 declare and initialize two arrays, x and y, to represent the vectors. Again, these declarations are "standard" implementations in that the user could call the dense BLAS on these arrays to perform, for instance, dot products or scalar-times-vector products ("axpy" operations in the BLAS terminology).

We create a tunable matrix object, A_tunable, from the input matrix by a call to **oski_CreateMatCSR** (lines 15–19) with the following arguments:

Listing 1: **A usage example without tuning**. This example illustrates basic object creation and kernel execution in our interface. Here, we perform one sparse matrix-vector multiply for a lower triangular matrix A with all ones on the diagonal, as shown in the leading comment.

```
// This example computes $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$, where
// $A = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix}$, $x = \begin{pmatrix} .25 \\ .45 \\ .65 \end{pmatrix}$, and $y$ is initially $\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$
// $A$ is a sparse lower triangular matrix with a unit diagonal, and $x, y$ are dense vectors.

// User's initial matrix and data
int Aptr[] = { 0, 0, 1, 2 };
int Aind[] = { 0, 0 };
double Aval[] = { −2, 0.5 };
double x[] = { .25, .45, .65 };
double y[] = { 1, 1, 1 };

double alpha = −1, beta = 1;

// Create a tunable sparse matrix object.
oski_matrix_t A_tunable = oski_CreateMatCSR(
    Aptr, Aind, Aval, 3, 3,  // CSR arrays
    SHARE_INPUTMAT,          // "copy mode"
    // remaining args specify how to interpret non-zero pattern
    3, INDEX_ZERO_BASED, MAT_TRI_LOWER, MAT_UNIT_DIAG_IMPLICIT );

// Create wrappers around the dense vectors.
oski_vecview_t x_view = oski_CreateVecView( x, 3, STRIDE_UNIT );
oski_vecview_t y_view = oski_CreateVecView( y, 3, STRIDE_UNIT );

// Perform matrix vector multiply, $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$.
oski_MatMult( A_tunable, OP_NORMAL, alpha, x_view, beta, y_view );

// Clean-up interface objects
oski_DestroyMat( A_tunable );
oski_DestroyVecView( x_view );
oski_DestroyVecView( y_view );

// Print result, y. Should be "[ .75 ; 1.05 ; .225 ]"
printf( "Answer: y = [ %f ; %f ; %f ]\n", y[0], y[1], y[2] );
```

1. Arguments 1–3 specify the CSR arrays (line 16).

2. Arguments 4–5 specify the matrix dimensions (line 16).

3. The 6th argument to **oski_CreateMatCSR** (line 17) specifies one of possible two *copy modes* for the matrix object, to help control the number of copies of the assembled matrix that may exist at any point in time. The value SHARE_INPUTMAT indicates that both the user and the library will share the CSR arrays Aptr, Aind, and Aval, because the user promises (a) not to free the arrays before destroying the object A_tunable via a call to **oski_DestroyMat** (line 29), and (b) to adhere to a particular set of read/write conventions. The other available mode, COPY_INPUTMAT, indicates that the library must make a copy of these arrays before returning from this call, because the user may choose to free the arrays at any time. We discuss the semantics of both modes in detail in Section 3 on page 13. In this example, the reader may regard A_tunable to be a wrapper around these arrays.

4. Arguments 7–10 tell the library how to interpret the CSR arrays (lines 19). Argument 7 is a count that says the next 3 arguments are semantic properties needed to interpret the input matrix correctly. First, INDEX_ZERO_BASED says that the index values in Aptr and Aind follow the C convention of starting at 0, as opposed to the typical Fortran convention of starting at 1 (the default is 1-based indexing if not otherwise specified). The value MAT_TRI_LOWER asserts the pattern is lower triangular and MAT_UNIT_DIAG_IMPLICIT asserts that no diagonal elements are specified explicitly but should be taken to be 1. The library may, at this call, check these properties to ensure they are true if the cost of doing so is $O(\text{no. of non-zeros})$.

Since this exapmle uses the SHARE_INPUTMAT copy mode and performs no tuning, the user can be sure A_tunable will not create any copies of the input matrix.

The routine **oski_CreateMatCSR** accepts a variable number of arguments; only the first 6 arguments are required. If the user does not provide the optional arguments, the library assumes the defaults discussed in Section 3.2 on page 14.

Dense vector objects of type oski_vecview_t, are always wrappers around user array representations (lines 22–23). We refer to such wrappers as *views*. A vector view encapsulates basic information about an array, such as its length, or such as the stride between consecutive elements of the vector within the array. As with the BLAS, a non-unit stride allows a dense vector to be a submatrix. In addition, an object of type oski_vecview_t can encapsulate multiple vectors (*multivector*) for kernels like sparse matrix-multiple vector multiply (SpMM) or triangular solve with multiple simultaneous right-hand sides. The multivector object would also store the number of vectors and the memory organization (*i.e.*, row vs. column major), as discussed Section 3.2.3 on page 16. Requiring the user to create a view in both the single- and multiple-vector cases helps unify and simplify some of the kernel argument lists.

The argument lists to kernels, such as **oski_MatMult** for SpMV in this example (line 26), follow some of the conventions of the dense BLAS. For example, a user can specify the constant OP_TRANS as the second argument to apply A^T instead of A, or specify other values for α and β.

The calls to **oski_DestroyMat** and **oski_DestroyVecView** free any memory allocated by the library to these objects (lines 29–31). However, since the user and library share the arrays underlying A_tunable, x_view, and y_view, the user is responsible for deallocating these arrays (here, Aptr, Aind, Aval, x, and y).

That A_tunable, x_view, and y_view are shared with the library implies the user can continue to operate on the data to which these views point as she normally would. For instance, the user can call dense BLAS operations, such as a dot products or scalar-vector multiply (the so-called "axpy" operation), on x and y, as shown in the biconjugate gradient example of Section 4 on page 28. Moreover, the user might choose to introduce calls to the OSKI kernels selectively, or gradually over time.

2.2 Providing explicit tuning hints

The user tunes a sparse matrix object by optionally providing one or more tuning "hints," followed by an explicit call to the matrix tuning routine, **oski_TuneMat**. Hints describe the expected workload, or assert performance-relevant structural properties of the matrix non-zeros.

Listing 2 on the next page sketches a simple example in which we provide two tuning hints. The first hint, made via a call to **oski_SetHintMatMult**, specifies the expected workload. We refer to such a hint as a *workload hint*. This example tells the library that the likely workload consists of at least a total of 500 SpMV operations on the same matrix. The argument list looks identical to the corresponding argument list for the kernel call, **oski_MatMult**, except that there is one additional parameter to specify the expected frequency of SpMV operations. The frequency allows the library to decide whether there are enough SpMV operations to hide the cost of tuning. For optimal tuning, the values of these parameters should match the actual calls as closely as possible.

The constant SYMBOLIC_VEC indicates that we will apply the matrix to a single vector with unit stride. Alternatively, we could use the constant SYMBOLIC_MULTIVEC to indicate that we will perform SpMM on at least two vectors. Better still, we could pass an actual instance of a oski_vecview_t object which has the precise stride and data layout information. Analagous routines exist for each of the other kernels in the system.

The second hint, made via a call to **oski_SetHint**, is a *structural hint* telling the library that we believe that the matrix non-zero structure is dominated by a single block size. Several of the possible structural hints accept optional arguments that may be used to qualify the hint—for this example, the user might explicitly specify a block size, though here she instead uses the constant ARGS_NONE to avoid doing so. The library implementation might then know to try register blocking since it would be most likely to yield the fastest implementation [28]. We describe a variety of other hints in Section 3.4 on page 19. These hints are directly related to candidate optimizations explored in our work, and we expect the list of hints to grow over time.

The actual tuning (*i.e.*, possible change in data structure) occurs at the call to **oski_TuneMat**. This example happens to execute SpMV exactly 500 times, though there is certainly no requirement to do so. Indeed, instead of specifying an exact number or estimate, the user may force the library to try a "moderate" level of tuning by specifying the symbolic constant ALWAYS_TUNE, or an "aggressive" level of tuning by specifying ALWAYS_TUNE_AGGRESSIVELY. The relative amount of tuning is implementation-dependent. These constants instruct the library to go ahead and try tuning at the next call to **oski_TuneMat**, assuming the application can always amortize cost. This facility might be useful when, say, benchmarking an application on a test problem to gauge the potential performance improvement from tuning.

Once A_tunable has been created, a user may call the tuning hints as often as and whenever she chooses. For example, suppose the user mixes calls to SpMV and $A^T A \cdot x$ in roughly equal proportion. The user can specify such a workload as follows:

```
oski_SetHintMatMult( A_tunable, ..., 1000 );
oski_SetHintMatTransMatMult( A_tunable, ..., 1000 );
// ... other hints ...
oski_TuneMat( A_tunable );
```

Listing 2: **An example of basic explicit tuning**. This example creates a sparse matrix object A_tunable and then tunes it for a workload in which we expect to call SpMV 500 times. In addition, we provide an additional hint to the library that the matrix non-zero structure is dominated by a dense blocks of a single size, uniformly aligned. Later in the application, we actually call SpMV a total of 500 times in some doubly nested loop.

```
// Create a tunable sparse matrix object.
A_tunable = oski_CreateMatCSR( ... );

// Tell the library we expect to perform 500 SpMV operations with α = 1, β = 1.
oski_SetHintMatMult( A_tunable, OP_NORMAL, 1.0, SYMBOLIC_VEC, 1.0, SYMBOLIC_VEC,
    500 ); // workload hint
oski_SetHint( A_tunable, HINT_SINGLE_BLOCKSIZE, ARGS_NONE ); // structural hint
oski_TuneMat( A_tunable );

// ...
{
    oski_vecview_t x_view = oski_CreateVecView( ... );
    oski_vecview_t y_view = oski_CreateVecView( ... );

    for( i = 0; i < 100; i++ ) {
        // ...
        for( k = 0; k < 5; k++ ) {
            // ...
            oski_MatMult( A_tunable, OP_NORMAL, 1.0, x_view, 1.0, y_view );
            // ...
        }
        // ...
    }
}
```

Listing 3: **An example of implicit tuning**. This example calls **oski_TuneMat** periodically, without explicitly providing any hints. At each call to **oski_TuneMat**, the library potentially knows more and more about how the user is using A_tunable and may therefore tune accordingly.

```
oski_matrix_t A_tunable = oski_CreateMatCSR( ... );
oski_vecview_t x_view = oski_CreateVecView( ... );
oski_vecview_t y_view =  oski_CreateVecView( ... );

oski_SetHint( A_tunable, HINT_SINGLE_BLOCKSIZE, 6, 6 );

// ...

for( i = 0; i < num_times; i++ ) {
    // ...
    while( !converged ) {
        // ...
        oski_MatMult( A_tunable, OP_NORMAL, 1.0, x_view, 1.0, y_view );
        // ...
    }
    oski_TuneMat( A_tunable );
    // ... maybe change a few non-zero values for the next solve ...
}
```

Then, **oski_TuneMat** will try to choose a data structure that yields good performance overall for this workload.

Workload hints are cumulative, *i.e.*, the call

```
oski_SetHintMatMult( A_tunable, ..., 2000 );
```

is equivalent to the two-call sequence

```
oski_SetHintMatMult( A_tunable, ..., 1000 );
oski_SetHintMatMult( A_tunable, ..., 1000 );
```

assuming the arguments given by "..." are identical, and furthermore independent of what other operations occur in between the two calls.

2.3 Tuning based on implicit profiling

Sparse matrix objects may also be tuned without any explicit hints. In this case, the library may quietly monitor the number of times each is called with a particular matrix and kernel arguments.

For instance, suppose that we cannot know statically the number of iterations that the innermost while loop executes in Listing 3. At run-time, the library implementation can log the calls to **oski_MatMult**, so that if and when the application calls **oski_TuneMat**, the library can make an educated guess about whether SpMV is called a sufficient number of times to hide the cost of tuning.

Hints may be ignored completely by the library, so the precise behavior when specifying hints, particularly if they are interleaved between executions of **oski_TuneMat**, cannot be precisely defined. We provide some guidelines in Section 3.4 on page 19.

3 Interface

The available library routines fall into 5 broad categories, summarized as follows:

1. **Creating and modifying sparse matrix and dense vector objects** (Section 3.2 on the next page; Table 1 on the following page): A sparse matrix object must be created from an existing user-allocated, *pre-assembled* matrix. We refer to this user-assembled matrix as the *input matrix*. (Appendix A on page 38 defines currently supported input matrix formats.) The user may specify whether the library and the user "share" the input matrix arrays (Section 3.2.1 on the following page). When the library "tunes" a matrix object, it may choose a new internal representation (sparse data structure).

   Dense vector objects are wrappers around user-allocated dense arrays.

2. **Executing kernels** (Section 3.3 on page 18; Table 5 on page 18), *e.g.*, sparse matrix-vector multiply, triangular solve: The interfaces to our kernel routines mimic the "look-and-feel" of the BLAS.

3. **Tuning** (Section 3.4 on page 19; Table 8 on page 20): Tuning occurs only if and when the user calls a particular routine in our interface. In addition to this "tune" routine, we also provide auxiliary routines that allow users to provide optional tuning hints.

4. **Saving and restoring tuning transformations** (Section 3.5 on page 24; Table 14 on page 27): We provide a routine to allow the user to see a precise description, represented by a string, of the transformations that convert the input matrix data structure to the tuned data structure.

   The user may then call an additional routine to "execute" this program on the same or similar input matrix, thereby providing a way to save and restore tuning transformations across application runs, in the spirit of FFTW's *wisdom* mechanism [20]. Moreover, the save/restore facility is an additional way for an advanced user to specify her own sequence of optimizing transformations.

   The interface itself does not define the format of these string-based transformations. However, we suggest a procedural, high-level scripting language, OSKI-Lua (derived from the Lua language [26]), for representing such transformations. We provide a high-level overview of OSKI-Lua in Section 5 on page 29.

5. **Error-handling** (Section 3.6 on page 24; Table 15 on page 27): In addition to the error codes and values returned by every routine in the interface, a user may optionally specify her own handler to be called when errors occur to access additional diagnostic information.

Tables 1–15 summarize the available routines. A user who only needs BLAS-like kernel functionality for her numerical algorithms or applications only needs to know about the object creation and kernel routines (Categories 1 and 2 above). Although tuning (*i.e.*, Categories 3 and 4) is an important part of our overall design, its use is strictly optional.

The C bindings are presented in detail in Appendix B on page 40. The following text provides an overview of the semantics and intent behind these bindings.

3.1 Basic scalar types

Most sparse matrix formats require storing both floating-point data for non-zero values and integer index data. Our interface is defined in terms of two scalar types accordingly:

Matrix objects	**oski_CreateMatCSR**	Create a valid, tunable matrix object from a CSR input matrix.
	oski_CreateMatCSC	Create a valid, tunable matrix object from a CSC input matrix.
	oski_CopyMat	Clone a matrix object.
	oski_DestroyMat	Free a matrix object.
	oski_GetMatEntry	Get the value of a specific matrix entry.
	oski_SetMatEntry	Set the value of a specific non-zero entry.
	oski_GetMatClique	Get a block of values, specified as a clique.
	oski_SetMatClique	Change a block of non-zero values specified as a clique.
	oski_GetMatDiagValues	Get values along a diagonal of a matrix.
	oski_SetMatDiagValues	Change values along a diagonal.
Vector objects	**oski_CreateVecView**	Create a view object for a single vector.
	oski_CreateMultiVecView	Create a view object for a multivector.
	oski_CopyVecView	Clone a vector view object.
	oski_DestroyVecView	Free a (multi)vector view object.

Table 1: **Creating and modifying matrix and vector objects**. Bindings appear in Appendix B.1 on page 40.

oski_value_t and oski_index_t. By default, these types are bound to double and int, respectively, but may be overridden at library build-time.

Our implementation of this interface also allows a user to generate, at library build-time, separate interfaces bound to other ordinal and value type combinations to support applications that need to use multiple types. These other interfaces are still C and Fortran callable, but the names are "mangled" to include the selected type information.

In some instances in which a value of type oski_value_t is returned, a NaN value is possible. Since oski_value_t may be bound to either a real or complex type, we denote NaN's by NaN_VALUE throughout.

3.2 Creating and modifying matrix and vector objects

Our interface defines two basic abstract data types for matrices and vectors: oski_matrix_t and oski_vecview_t, respectively. Available primitives to create and manipulate objects of these types appears in Table 1, and C bindings appear in Appendix B.1 on page 40.

3.2.1 Creating matrix objects

The user creates a matrix object of type oski_matrix_t from a valid input matrix. Logically, such an object represents at most one copy of a user's input matrix tuned for some kernel workload, with a fixed non-zero pattern for the entire lifetime of the object.

At present, we support 0- and 1-based CSR and CSC representations for the input matrix. For detailed definitions of valid input formats, refer to Appendix A on page 38.

All of the supported input matrix formats use array representations, and a typical call to create a matrix object from, say, CSR format looks like

A_tunable = **oski_CreateMatCSR**(Aptr, Aind, Aval, num_rows, num_cols, <copy mode>,
$\langle k \rangle$, $\langle property\_1 \rangle$, ..., $\langle property\_k \rangle$);

where A_tunable is the newly created matrix object, Aptr, Aind, and Aval are user created arrays that store the input matrix (here in a valid CSR format), <copy mode> specifies how the library should copy the input matrix data, and <*property_1*> through <*property_k*> specify how the library should interpret that data.

To make memory usage logically explicit, the interface supports two data *copy modes*. These modes, defined by the scalar type oski_copymode_t (Table 2 on the following page), are:

1. COPY_INPUTMAT: The library makes a copy of the input matrix arrays, Aptr, Aind, and Aval. The user may modify or free any of these arrays after the return from **oski_CreateMat** without affecting the matrix object A_tunable. Similarly, any changes to the matrix object do not affect any of the input matrix arrays.

   If the user does not or cannot free the input matrix arrays, then two copies of the matrix will exist.

2. SHARE_INPUTMAT: The user and the library agree to *share* the input matrix arrays subject to the following conditions:

   (a) The user promises that the input matrix arrays will not be freed or reallocated before a call to **oski_DestroyMat** on the handle A_tunable.

   (b) The user promises not to modify the elements of the input matrix arrays *except* through the interface's set-value routines (Section 3.2.2 on the next page). This condition helps the library keep any of its internally maintained, tuned copies consistent with the input matrix data.

   (c) The library promises not to change any of the values in Aptr, Aind. This condition fixes the pattern and maintains the properties of the input matrix data given at creation time. Elements of Aval may change only on calls to the set-value routines.

   (d) The library promises to keep the input matrix arrays and any tuned copies synchronized. This condition allows the user to continue to read these arrays as needed. That is, if the user calls a set-value routine to change a non-zero value, the library will update its internal tuned copy (if any) *and* the corresponding non-zero value stored in the input matrix array Aval.

   The significance of this shared mode is that, in the absence of explicit calls to the tuning routine, only one copy of the matrix will exist, *i.e.*, the user may consider A_tunable to be a wrapper or view of the input matrix.

Properties (<*property_1*> through <*property_k*> in this example) are optional, and the user should specify as many as needed for the library to interpret the non-zero pattern correctly. For instance, Listing 1 on page 8 creates a matrix with implicit ones on the diagonal which are not stored, so the user must specify MAT_UNIT_DIAG_IMPLICIT as a property. A list of available properties appears in Table 3 on page 17, where default properties assumed by the library are marked with a red asterisk (*).

The user may create a copy of A_tunable by calling **oski_CopyMat**. This copy is logically equivalent to creating a matrix object in the COPY_BUFFERS mode. The user frees A_tunable or its copies by a call to **oski_DestroyMat**.

In addition to user-created matrix objects, there is one immutable pre-defined matrix object with a special meaning: INVALID_MAT. This matrix is returned when matrix creation fails, and is conceptually a constant analogous to the NULL constant for pointers in C.

SHARE_INPUTMAT	User and library agree to share the input matrix arrays
COPY_INPUTMAT	The library copies the input matrix arrays, and the user may free them immediately upon return from the handle creation routine.

Table 2: **Copy modes (type oski_copymode_t).** Copy modes for the matrix creation routines, as discussed in detail in Section 3.2.1 on page 14.

3.2.2 Changing matrix non-zero values

The non-zero pattern of the input matrix fixes the non-zero pattern of A_tunable, but the user may modify the non-zero values. If the input matrix contains explicit zeros, the library treats these entries as *logical non-zeros* whose values may be modified later. We provide several routines to change non-zero values. To change individual entries, the user may call **oski_SetMatEntry**, and to change a block of values defined by a a clique, the user may call **oski_SetMatClique**. If A_tunable is a shallow copy of the user's matrix, the user's values array will also change. Logical non-zero values are subject to properties asserted at matrix creation-time (see Appendix B.1 on page 40).

We also define primitives for obtaining all of the values along an arbitrary diagonal and storing them into a dense array (**oski_GetMatDiagValues**), and for setting all of the non-zero values along an arbitrary diagonal from a dense array (**oski_SetMatDiagValues**). The same restriction on altering only non-zero values in the original matrix applies for these routines.

Tuning may select a new data structure in which explicit zero entries are stored that were implicitly 0 (*i.e.*, not stored) in the input matrix. The behavior if the user tries to change these entries is not defined, for two reasons. First, allowing the user to change these entries would yield inconsistent behavior across platforms for the same matrix, since whether a "filled-in" entry could be changed would depend on what data structure the library chooses. Second, requiring that the library detect all such attempts to change these entries might, in the worst case, require keeping a copy of the original input matrix pattern, creating memory overhead. The specifications in Appendix B.1 on page 40 allow, but do not require, the library implementation to report attempts to change implicit zeros to non-zero values as errors.

3.2.3 Vector objects

Vector objects (type oski_vecview_t) are always views on the user's dense array data. Such objects may be views of either single column vectors, created by a call to **oski_CreateVecView**, or multiple column vectors (multivectors), created by a call to **oski_CreateMultiVecView**. A multivector consisting of $k \geq 1$ vectors of length n each is just a dense $n \times k$ matrix, but we use the term multivector to suggest a common case in applications in which k is on the order of a "small" constant (*e.g.*, 10 or less). A single vector is the same as the multivector with $k = 1$.

This interface expects the user to store her multivector data as a dense matrix in either *row major* (C default) or *column major* (Fortran default) array storage (Table 4 on page 18). The interface also supports submatrices by allowing the user to provide the *leading dimension* (or *stride*), as is possible with the dense BLAS. Thus, users who need the BLAS can continue to mix BLAS operations on their data with calls to the OSKI kernels.

In addition to user-created vector views, we define two special, immutable vector view objects: SYMBOLIC_VEC and SYMBOLIC_MULTIVEC. Conceptually, these objects are

*MAT_GENERAL	Input matrix specifies all non-zeros.
MAT_TRI_UPPER	Only non-zeros in the upper triangle exist.
MAT_TRI_LOWER	Only non-zeros in the lower triangle exist.
MAT_SYMM_UPPER	Matrix is symmetric but only the upper triangle is stored.
MAT_SYMM_LOWER	Matrix is symmetric but only the lower triangle is stored.
MAT_SYMM_FULL	Matrix is symmetric and all non-zeros are stored.
MAT_HERM_UPPER	Matrix is Hermitian but only the upper triangle is stored.
MAT_HERM_LOWER	Matrix is Hermitian but only the lower triangle is stored.
MAT_HERM_FULL	Matrix is Hermitian and all non-zeros are stored.
*MAT_DIAG_EXPLICIT	Any non-zero diagonal entries are specified explicitly.
MAT_UNIT_DIAG_IMPLICIT	No diagonal entries are stored, but should be assumed to be equal to 1.
*INDEX_ONE_BASED	Array indices start at 1 (default Fortran convention).
INDEX_ZERO_BASED	Array indices start at 0 (default C convention).
*INDEX_UNSORTED	Non-zero indices in CSR (CSC) format within each row (column) appear in any order.
INDEX_SORTED	Non-zero indices in CSR (CSC) format within each row (column) are sorted in increasing order.
*INDEX_REPEATED	Indices may appear multiple times.
INDEX_UNIQUE	Indices are unique.

Table 3: **Input matrix properties (type oski_inmatprop_t).** Upon the call to create a matrix object, the user may characterize the input matrix by specifying one or more of the above properties. Properties grouped within the same box are mutually exclusive. Default properties marked by a red asterisk (*).

LAYOUT_ROWMAJ	The multivector is stored in row-major format (as in C/C++).
LAYOUT_COLMAJ	The multivector is stored in column-major format (as in Fortran).

Table 4: **Dense multivector (dense matrix) storage modes (type oski_storage_t).** Storage modes for the dense multivector creation routines.

oski_MatMult	Sparse matrix-vector multiply (SpMV) $y \leftarrow \alpha \cdot \text{op}(A) \cdot x$ where $\text{op}(A) \in \{A, A^T, A^H\}$.
oski_MatTrisolve	Sparse triangular solve (SpTS) $x \leftarrow \alpha \cdot \text{op}(A)^{-1} \cdot x$
oski_MatTransMatMult	$y \leftarrow \alpha \cdot \text{op}_2(A) \cdot x + \beta \cdot y$ where $\text{op}_2(A) \in \{A^T A, A^H A, AA^T, AA^H\}$
oski_MatMultAndMatTransMult	Simultaneous computation of $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$ AND $z \leftarrow \omega \cdot \text{op}(A) \cdot w + \zeta \cdot z$
oski_MatPowMult	Matrix power multiplication Computes $y \leftarrow \alpha \cdot \text{op}(A)^\rho \cdot x + \beta \cdot y$

Table 5: **Sparse kernels.** This table summarizes all of the available sparse kernel routines. The user selects single or multivector versions by passing in an appropriate vector view (Section 3.2.3 on page 16). See Appendix B.3 on page 52 for bindings.

constants that may be used with the tuning workload specification routines to indicate tuning for single vectors or multivectors instead of specifying instantiated view objects. See Section 3.4 on the next page.

3.3 Executing kernels

We summarize the available kernels in Table 5, and present their bindings in Appendix B.3 on page 52. In addition to both single vector and multivector versions of sparse matrix-vector multiply (**oski_MatMult**) and sparse triangular solve (**oski_MatTrisolve**), we provide interfaces for three "high-level" sparse kernels that provide more opportunities to reuse the elements of A:

- Simultaneous multiplication of A and A^T (or A^H) by a dense multivector (**oski_MatMultAndMatTransMult**).
- Multiplication of $A^T \cdot A$ or $A \cdot A^T$ (or conjugate transpose variants) by a dense multivector (**oski_MatTransMatMult**).
- Multiplication of a non-negative integer power of a matrix (**oski_MatPowMult**).

We have recently reported on experimental justifications and suggested implementations for these kernels [47, 49].

3.3.1 Applying the transpose of a matrix

We follow the BLAS convention of allowing the user to apply the transpose (or, for complex data, the transpose or Hermitian transpose). See Table 6 on the following page for a list of

OP_NORMAL	Apply A.
OP_TRANS	Apply A^T.
OP_CONJ_TRANS	Apply $A^H = \bar{A}^T$, the conjugate transpose of A.

Table 6: **Matrix transpose options (type oski_matop_t).** Constants that allow a user to apply a matrix A, its transpose A^T, or, for complex-valued matrices, its conjugate transpose A^H. These options are called $op(A)$ in Table 5 on the preceding page.

OP_AT_A	Apply $A^T A$.
OP_AH_A	Apply $A^H A$.
OP_A_AT	Apply AA^T.
OP_A_AH	Apply AA^H.

Table 7: **Matrix-transpose-times-matrix options (type oski_ataop_t).** Constants that allow a user to apply $A^T A$, $A^H A$, AA^T, or AA^H in calls to the routine, **oski_MatTransMatMult**. These options are called $op_2(A)$ in Table 5 on the page before.

transpose options provided by the scalar type oski_matop_t. The notation op(A) indicates that any of A, A^T, or A^H may be applied, *i.e.*, op$(A) \in \{A, A^T, A^H\}$.

The high-level kernel **oski_MatTransMatMult** has inherently more matrix reuse opportunities. This kernel allows the user to apply any of the four matrix operations listed in Table 7, given a matrix A: AA^T, $A^T A$, AA^H, and $A^H A$.

3.3.2 Aliasing

The interface guarantees correct results only if multivector view object *input* arguments do not alias any multivector view object *output* arguments, *i.e.*, if input and output views do not view the same user data. If such aliasing occurs, the results are not defined.

3.3.3 Scalars vs. 1x1 matrix objects

An object of type oski_matrix_t created with dimensions 1×1 is *not* treated as a scalar by the kernel routines. Therefore, such an object may only be applied to a single vector and not a $n \times k$ multivector object when $k \geq 2$.

3.3.4 Compatible dimensions for matrix multiplication

All of the kernels apply a matrix op(A) to a (multi)vector x and store the result in another (multi)vector y. Let $m \times n$ be the dimensions of op(A), let $p \times k$ be the dimensions of x, and let $q \times l$ be the dimensions of y. We say these dimensions are *compatible* if $m = q$, $n = p$, and $k = l$.

3.3.5 Floating point exceptions

None of the kernels attempt to detect or to trap floating point exceptions.

3.4 Tuning

The user tunes a valid matrix object at any time and as frequently as she desires for a given matrix object of type oski_matrix_t. The library tunes by selecting a data structure

oski_SetHintMatMult **oski_SetHintMatTrisolve** **oski_SetHintMatTransMatMult** **oski_SetHintMatMult_and_MatTransMult** **oski_SetHintMatPowMult**	Workload hints specify the expected options and frequency of the corresponding kernel call.
oski_SetHint	Specify hints about the non-zero structure that may be relevant to tuning. For a list of available hints, see Table 9 on the following page.
oski_TuneMat	Tune the matrix data structure using all hints and implicit workload data accumulated so far.

Table 8: **Tuning primitives**. The user tunes a matrix object by first specifying workload and structural hints, followed by an explicit call to the tuning routine, **oski_TuneMat**. Bindings appear in Appendix B.4 on page 57.

customized for the user's matrix, kernel workload, and machine.[4] The interface defines three groups of tuning operations, listed in Table 8 and summarized as follows:

- **Workload specification** (Section 3.4.1 on page 22): These primitives allow the user to specify which kernels she will execute and how frequently she expects to execute each one. There is one workload specification routine per kernel.

- **Structural hint specification** (Section 3.4.2 on page 22): The user may optionally influence the tuning decisions by providing hints about the non-zero structure of the matrix. For example, the user may tell the library that she believes the structure of the matrix consists predominantly of uniformly aligned 6×6 dense blocks.

- **Explicit tuning** (Section 3.4.3 on page 23): The user must explicitly call the "tune routine," **oski_TuneMat**, to initiate tuning. Conceptually, this routine marks the point in program execution at which the library may spend time changing the data structure. The tune routine uses any hints about the non-zero structure or workload, whether they are specified explicitly by the user via calls to the above tuning primitives or they are gathered implicitly during any kernel calls made during the lifetime of the matrix object.

Section 2 on page 6 illustrates the common ways in which we expect users to use the interface to tune.

The library may optimize kernel performance by permuting the rows and columns of the matrix to reduce the bandwidth [27, 44, 13, 23, 14] or to create dense block structure [36]. That is, the library may compute a tuned matrix representation $\hat{A} = P_r \cdot A \cdot P_c^T$ for the user's matrix A, where P_r and P_c are permutation matrices. However, this optimization requires each kernel to permute its vectors on entry and exit to maintain the correctness of the interfaces. Section 3.4.4 on page 23 discusses functionality that allows the user, if she so desires, to determine if reordering has taken place and access P_r, P_c^T, and $\hat{A}$ directly to reduce the number of permutations.

[4]The interface also permits an implementation of this interface to generate code or perform other instruction-level tuning at run-time as well.

	Hint	Arguments	Description
1	HINT_NO_BLOCKS	none	Matrix contains little or no dense block substructure.
	HINT_SINGLE_BLOCKSIZE	[int r, c]	Matrix structure is dominated by a single block size, $r \times c$.
	HINT_MULTIPLE_BLOCKSIZES	[int k, r_1, c_1, $\ldots$, r_k, c_k]	Matrix structure consists of at least $k \geq 1$ multiple block sizes. These sizes include $r_1 \times c_1, \ldots, r_k \times c_k$.
2	HINT_ALIGNED_BLOCKS	none	Any dense blocks are uniformly aligned. That is, let (i, j) be the $(1, 1)$ element of a block of size $r \times c$. Then, $(i-1) \mod r = (j-1) \mod c = 0$.
	HINT_UNALIGNED_BLOCKS	none	Any dense blocks are not aligned, or the alignment is unknown.
3	HINT_SYMM_PATTERN	none	The matrix non-zero pattern is structurally symmetric, or nearly so.
	HINT_NONSYMM_PATTERN	none	The matrix non-zero pattern is structurally "very" unsymmetric.
4	HINT_RANDOM_PATTERN	none	The matrix non-zeros (or non-zero blocks) are nearly distributed uniformly randomly over all positions.
	HINT_CORRELATED_PATTERN	none	The row indices and column indices for non-zeros are highly correlated.
5	HINT_NO_DIAGS	none	The matrix contains little if any explicit diagonal structure.
	HINT_DIAGS	[int k, d_1, $\ldots$, d_k]	The matrix has structure best represented by multiple diagonals. The diagonal lengths include $d_1, \ldots, d_k$, possibly among other lengths.

Table 9: **Available structural hints (type oski_tunehint_t).** The user may provide additional hints, via a call to the routine **oski_SetHint**, about the non-zero structure of the matrix which may be useful to tuning. Several of the hints allow the user to specify additional arguments, shown in column 2. All arguments are optional. The table groups hints into 5 mutually exclusive sets, *e.g.*, a user should only specify one of HINT_NO_BLOCKS, HINT_SINGLE_BLOCKSIZE, and HINT_MULTIPLE_BLOCKSIZES if she specifies any of these hints at all.

ALWAYS_TUNE	The user expects "many" calls, and the library may therefore elect to do some basic tuning.
ALWAYS_TUNE_AGGRESSIVELY	The user expects a sufficient number of calls that the library may tune aggressively.

Table 10: **Symbolic calling frequency constants (type int).** Instead of providing a numerical estimate of the number of calls the user expects to make when specifying a workload hint, the user may use one of the above symbolic constants.

SYMBOLIC_VEC	A symbolic single vector view.
SYMBOLIC_MULTIVEC	A symbolic multivector view consisting of at least two vectors.

Table 11: **Symbolic vector views for workload hints (type oski_vecview_t).** Instead of passing an actual vector view object to the workload hint routine (Table 8 on page 20), the user may pass in one of the above symbolic views.

3.4.1 Providing workload hints explicitly

Each of the kernels listed in Table 5 on page 18 has a corresponding workload hint routine. The user calls these routines to tell the library which kernels she will call and with what arguments for a given matrix object, and the expected frequency of such calls. The routines for specifying workload hints (Table 8 on page 20) all have an argument signature of the form

oski_SetHint<KERNEL>(**A_tunable**, <KERNEL_PARAMS>, num_calls);

where num_calls is an integer. This hint tells the library that we will call the specified <KERNEL> on the object A_tunable with the arguments <KERNEL_PARAMS>, and that we expect to make num_calls such calls.

Instead of specifying an estimate of the number of calls explicitly, the user may substitute the symbolic constant ALWAYS_TUNE or ALWAYS_TUNE_AGGRESSIVELY to tell the library to go ahead and assume the application can amortize cost (see Table 10). The use of two constants allows a library implementation to provide two levels of tuning when the user cannot estimate the number of calls.

Where a kernel expects a vector view object to be passed as an argument, the user may pass to the workload hint either SYMBOLIC_VEC or SYMBOLIC_MULTIVEC instead of an actual vector view object (Table 11). The user should use SYMBOLIC_VEC if she anticipates using a single vector, or SYMBOLIC_MULTIVEC if she anticipates using at least two vectors. Specifying actual vector view objects is preferred since they will contain additional information relevant to tuning, including storage layout for multivectors (*i.e.*, row vs. column major) and strides or leading dimensions.

3.4.2 Providing structural hints

A user provides one or more structural hints by calling **oski_SetHint** as illustrated in Sections 2.2–2.3. Providing these hints is entirely optional, but a library implementation may use these hints to constrain a tuning search.

Some hints allow the user to provide additional information. For instance, consider the hint, HINT_SINGLE_BLOCKSIZE, which tells the library that the matrix structure is dominated by dense blocks of a particular size. Rather than just indicate the presence of a single block size by the following call

TUNESTAT_NEW	The library selected a new data structure for the matrix based on the current workload data and hints.
TUNESTAT_AS_IS	The library did not change the data structure.

Table 12: **Tuning status codes**. Status codes returned by **oski_TuneMat** in the event that no error occurred during tuning.

oski_SetHint(**A_tunable**, HINT_SINGLE_BLOCKSIZE, ARGS_NONE);

the user may specify the block size explicitly if it is known:

oski_SetHint(**A_tunable**, HINT_SINGLE_BLOCKSIZE, 6, 6); // 6×6 blocks

In this case, either call is "correct" since specifying the block size is optional. See Table 9 on page 21 for a list of hints, their arguments, and whether the arguments are optional or required.

A library implementation is free to ignore all hints, so there is no strict definition of the library's behavior if, for example, the user provides conflicting hints. We recommend that implementations use the following interpretation of multiple hints:

- If more than one hint from a mutually exclusive group is specified, assume the latter is true. For example, if the user specifies HINT_SINGLE_BLOCKSIZE followed by HINT_NO_BLOCKS, then no-block hint should override the single-block size hint.

- Hints from different groups should be joined by a logical 'and.' That is, if the user specifies HINT_SINGLE_BLOCKSIZE and HINT_SYMM_PATTERN, this combination should be interpreted as the user claiming the matrix is both nearly structurally symmetric and dominated by a single block size.

3.4.3 Initiating tuning

This interface defines a single routine, **oski_TuneMat**, which marks the point in program execution at which tuning may occur. As discussed in its binding (Section B.4 on page 57), **oski_TuneMat** returns one of the integer status codes shown in Table 12 to indicate whether it changed the data structure (TUNESTAT_NEW) or not (TUNESTAT_AS_IS).

3.4.4 Accessing the permuted form

The interface defines several routines (Table 13 on the following page) that allow the user to determine whether the library has optimized kernel performance by reordering the rows and columns of the matrix (by calling **oski_IsMatPermuted**), to extract the corresponding permutations (**oski_ViewPermutedMat**, **oski_ViewPermutedMatRowPerm**, **oski_ViewPermutedMatColPerm**), and to apply these permutations to vector view objects (**oski_PermuteVecView**).

Given the user's matrix A, suppose that tuning produces the representation $A = P_r^T \cdot \hat{A} \cdot P_c$, where P_r and P_c are permutation matrices and multiplying by $\hat{A}$ is much faster than multiplying by A. This form corresponds to reordering the rows and columns of A—by pre- and post-multiplying A by P_r and P_c^T—to produce an optimized matrix $\hat{A}$. To compute $y \leftarrow A \cdot x$ correctly while maintaining its interface and taking advantage of fast multiplication by $\hat{A}$, the kernel **oski_MatMult** must instead compute $P_r \cdot y \leftarrow \hat{A} \cdot (P_c \cdot x)$. Carrying out this computation in-place requires permuting x and y on entry, and then

oski_IsMatPermuted	Determine whether a matrix has been tuned by reordering its rows or columns.
oski_ViewPermutedMat	Returns a read-only matrix object for the re-ordered copy of A, $\hat{A}$.
oski_ViewPermutedMatRowPerm	Returns the row permutation P_r.
oski_ViewPermutedMatColPerm	Returns the column permutation P_c.
oski_PermuteVecView	Apply a permutation object (or its inverse/-transpose) to a vector view.

Table 13: **Extracting and applying permuted forms**. If tuning produces a tuned matrix $\hat{A} = P_r \cdot A \cdot P_c^T$, the above routines allow the user to detect and to extract read-only views of P_r, P_c, and $\hat{A}$, and apply the permutations P_r and P_c. Bindings appear in Appendix B.5 on page 61.

again on return. If tuning produces such a permuted matrix, all kernels perform these permutations as necessary.

Since the user may be able to reduce the permutation cost by permuting only once before executing her algorithm, and perhaps again after her algorithm completes, we provide several routines to extract *view objects* of P_r, P_c, and $\hat{A}$. These objects are views of the internal tuned representation of A. Therefore, they are valid for the lifetime of the matrix object that represents A, they do not have to be deallocated explicitly by the user. Moreover, if A is re-tuned, these representations will be updated automatically.

We provide an additional type for permutations, oski_perm_t, and the routines listed in Table 13. An object of type oski_perm_t may equal a special symbolic constant representing an identity permutation of any size, PERM_IDENTITY. This constant may be used in either of the routines to apply a permutation or its inverse to a vector view.

Listing 4 on the following page sketches the way in which a user might use these routines in her application. It reduces the number of permutations performed if $P_r = P_c$, a condition easily tested (line 20) by directly comparing the corresponding permutation variables **Pr** and **Pc**.

3.5 Saving and restoring tuning transformations

The interface defines basic facilities that allow users to view the tuning transformations which have been applied to a matrix, to edit these transformations, and to re-apply them (Table 14 on page 27). To get a string describing how the input matrix data structure was transformed during tuning, the user calls **oski_GetMatTransforms**. This routine returns a newly allocated string containing the transformations description. To set the data structure (*i.e.*, to apply some set of transformations to the input matrix data structure), the user calls **oski_ApplyMatTransforms**. Such a call is equivalent to calling **oski_TuneMat**, except that instead of allowing the library to decide what data structure to use, we are specifying it explicitly. We illustrate the usage of these two routines in Listing 5 on page 26 and Listing 6 on page 26.

We do not mandate the precise format of the string, but strongly encourage the use of a human-readable format. Section 5 on page 29 provides several examples of transformations exprssed in a high-level procedural language based on Lua [26].

3.6 Handling errors

The OSKI interface provides two methods for handling errors:

Listing 4: **An example of extracting permutations**. This example computes $y \leftarrow A^k \cdot x$. Suppose the library tunes by symmetrically reordering the rows and columns, *i.e.*, by computing $A = P^T \cdot \hat{A} \cdot P$ where P is a permutation matrix and multiplying by $\hat{A}$ is "much faster" than multiplying by A. Then, this example shows how to pre- and post-permute x, y only each, instead of at every multiplication $A \cdot x$.

```
// Create A, x, and y.
oski_matrix_t A_tunable = ...;
oski_vecview_t x_view = ..., y_view = ...;

// Store permuted form. Declared as 'const' since they will be read-only.
const oski_perm_t Pr, Pc;        // Stores $P_r, P_c$
const oski_matrix_t A_fast;      // Stores $\hat{A}$

int iter, max_power = 3; // $k$

// Tune for our computation
oski_SetHintMatMult( A_tunable, OP_NORMAL, 1, x_view, 1, y_view, max_power );
oski_TuneMat( A_tunable );

// Extract permuted form, $\hat{A} = P_r \cdot A \cdot P_c^T$
A_fast = oski_ViewPermutedMat( A_tunable );
Pr = oski_ViewPermutedMatRowPerm( A_tunable );
Pc = oski_ViewPermutedMatColPerm( A_tunable );

if ( Pr == Pc ) // May reduce the number of permutations needed?
{
    // Compute $y \leftarrow A^k \cdot x$ in three steps.
    // 1. $y \leftarrow P_r \cdot y$, $x \leftarrow P_c \cdot x$
    oski_PermuteVecView( Pr, OP_NORMAL, y_view );
    oski_PermuteVecView( Pc, OP_NORMAL, x_view );

    // 2. $y \leftarrow A^k \cdot x$
    for( iter = 0; iter < max_power; iter++ )
        oski_MatMult( A_fast, OP_NORMAL, 1.0, x_view, 1.0, y_view );

    // 3. $y \leftarrow P_r^T \cdot y$, $x \leftarrow P_c^T \cdot x$
    oski_PermuteVecView( Pr, OP_TRANS, y_view );
    oski_PermuteVecView( Pc, OP_TRANS, x_view );
}
else // use a conventional implementation
    for( iter = 0; iter < max_power; iter++ )
        oski_MatMult( A_tunable, OP_NORMAL, 1.0, x_view, 1.0, y_view );

// Clean-up
oski_DestroyMat( A_tunable ); // Invalidates Pr, Pc, and A_fast
oski_DestroyVecView( x_view ); oski_DestroyVecView( y_view );
```

Listing 5: **An example of saving transformations**. This example extracts the tuning transformations applied to a matrix object A_tunable and saves them to a file.

```
oski_matrix_t A_tunable = oski_CreateMatCSR( . . . );

char* xforms;  // stores transformations
FILE* fp_saved_xforms = fopen( "./my_xform.txt", "wt" ); // text file for output

// ...

// Tune the matrix object
oski_TuneMat( A_tunable );

// ...

// Extract transformations
xforms = oski_GetMatTransforms( A_tunable );
printf( "--- Matrix transformations ---\n%s\n--- end ---\n", xforms );

// Save to a file
fprintf( fp_saved_xforms, "%s\n", xforms );
fclose( fp_saved_xforms );

free( xforms );

// ...
```

Listing 6: **An example of applying transformations**. This example reads a string representation of tuning transformations from a file and applies them to an untuned matrix.

```
FILE* fp_saved_xforms = fopen( "./my_xform.txt", "rt" ); // text file for input

// Buffer for storing transformation read from the file. The actual size of this buffer should
// should be the file size, but we use some maximum constant here for illustration purposes.
char xforms[ SOME_MAX_BUFFER_SIZE ];
int num_chars_read;

oski_matrix_t A_tunable = oski_CreateMat_CSR( . . . );

// Read transformations.
num_chars_read = fread( xforms, sizeof(char), SOME_MAX_BUFFER_SIZE−1, fp_saved_xforms );
xforms[num_chars_read] = NULL;

// Execute one un-tuned matrix multiply.
oski_MatMult( A_tunable, . . . );

// Change matrix data structure.
oski_ApplyMatTransforms( A_tunable, xforms );

// Now perform matrix multiply in the new data structure.
oski_MatMult( A_tunable, . . . );

// . . .
```

oski_GetMatTransforms	Retrieve a string representation of the tuning transformations that have been applied to a given matrix.
oski_ApplyMatTransforms	Apply tuning transformations to a given matrix.

Table 14: **Saving and restoring tuning transformations**. The interface defines a basic facility to allow users to view the tuning transformations that have been applied to matrix, and later re-apply those (or other) transformations to another matrix. Bindings appear in Appendix B.6 on page 63.

oski_GetErrorHandler	Returns a pointer to the current error handling routine.
oski_SetErrorHandler	Changes the current error handling routine to a user-supplied handler.
oski_HandleErrorDefault	The default error handler provided by the library.

Table 15: **Error handling routines**. Bindings appear in Appendix B.7 on page 64.

1. **Error code returns**: Many of the routines in the interface return an integer error code (of type int). All of the possible error codes are negative, providing a simple way for an application to check for an error on return from any OSKI routine.
2. **Error handling routines**: The library calls an error handling routine when an error occurs. By default, this routine is **oski_HandleErrorDefault**. However, the user may also replace this routine with her own handler, or replace it with NULL to disable error handler calling entirely.

When an error condition is detected within a OSKI routine, it is *always* handled using the following procedure:

- The routine calls the current error handler.
- Regardless of which error handler is called (if any), the routine may return an error code.

A user may change the error handler by calling **oski_SetErrorHandler**, or retrieve the current error handler by calling **oski_GetErrorHandler**. The error handling routines are summarized in Table 15.

An error handling routine has the following signature (oski_errhandler_t):

```
void your_handler( int error_code, const char* message,
    const char* source_filename, unsigned long line_number,
    const char* format_string, ... );
```

The first 4 parameters describe the error and its source location. The arguments beginning at **format_string** are printf-compatible arguments that provide a flexible way to provide supplemental error information.

For example, the following code shows how the error handler might be called from within the the SpMV kernel, **oski_MatMult**, when the user incorrectly tries to apply a matrix A_tunable with dimensions m×n to a vector of length veclen, where $n \neq$ veclen:

```
if( n ≠veclen ) {
    your_handler( ERR_DIM_MISMATCH, "oski_MatMult: Mismatched dimensions",
        "oski_MatMult.c", 507,
        "Matrix dimensions: %d x %d, Vector length: %d\n", m, n, veclen );
    return ERR_DIM_MISMATCH;
}
```

4 Example: Biconjugate Gradients

We present a subroutine implementation of the biconjugate gradient (BiCG) algorithm (without preconditioning) for solving a system of linear equations [40, Chapter 7, p. 223]. This example mixes calls to OSKI and the dense BLAS.

```
// Solves the n × n system A · x = b for x using the BiCG algorithm.
// The vector x should be initialized with a starting guess.
int SolveLinSys_using_BiCG( const oski_matrix_t A, int n, const double* b, double* x )
{
    int converged = 0;  // == 1 when algorithm has converged.
    double *workspace; // Temporary vector storage space.
    double *p, *ps, *y, *ys, *r, *rs;   // Temporary vectors of length n each
    oski_vecview_t p_view, ps_view, y_view, ys_view, r_view, x_view; // For OSKI

    // Argument error checking
    assert( n > 0 );
    assert( A ≠INVALID_MAT );
    assert( x ≠NULL );
    assert( b ≠NULL );

    // Allocate 6 temporary vectors as a block.
    workspace = (double *)malloc( sizeof(double) * n * 6 );
    assert( workspace ≠NULL );

    p = workspace; ps = workspace + n;
    y = workspace + 2*n; ys = workspace + 3*n;
    r = workspace + 4*n; rs = workspace + 5*n;

    p_view = oski_CreateVecView( p, n, STRIDE_UNIT );
    ps_view = oski_CreateVecView( ps, n, STRIDE_UNIT );
    y_view = oski_CreateVecView( y, n, STRIDE_UNIT );
    ys_view = oski_CreateVecView( ys, n, STRIDE_UNIT );
    r_view = oski_CreateVecView( r, n, STRIDE_UNIT );
    rs_view = oski_CreateVecView( rs, n, STRIDE_UNIT );

    // Compute residual r_0 ← b, r_0 ← r_0 − A · x_0
    dcopy( n, b, 1, r, 1 );
    oski_MatMult( A, OP_NORMAL, −1.0, x_view, 1.0, r_view );

    dcopy( n, r, 1, rs, 1 );   // r_0^* ← r
    dcopy( n, r, 1, p, 1 );    // p_0 ← r
    dcopy( n, r, 1, ps, 1 );   // p_0^* ← r

    while( !converged ) {
        // Inner loop, iteration j (starting at j = 0)
        double r_dot_rs, alpha, beta;

        // Simultaneously compute: y ← A · p_j, y^* ← A^T · p_j^*
        oski_MatMultAndMatTransMult( A, 1.0, p_view, 0.0, y_view,
            OP_TRANS, 1.0, ps_view, 0.0, ys_view );
        // $\alpha_j \leftarrow (r_j^T \cdot r_j^*) / ((A \cdot p_j)^T \cdot p_j^*)$
        r_dot_rs = ddot( n, r, 1, rs, 1 );
        alpha = r_dot_rs / ddot(n, y, 1, ps, 1);

        daxpy( n, alpha, p, 1, x, 1 ); // $x_{j+1} \leftarrow x_j + \alpha_j \cdot p_j$
        daxpy( n, −alpha, y, 1, r, 1 ); // $r_{j+1} \leftarrow r_j - \alpha_j \cdot A \cdot p_j$
        daxpy( n, −alpha, ys, 1, rs, 1 ); // $r_{j+1}^* \leftarrow r_j^* - \alpha_j \cdot A^T \cdot p_j^*$
        beta = ddot(n, r, 1, rs, 1) / r_dot_rs; // $\beta_j \leftarrow (r_{j+1}^T \cdot r_{j+1}^*) / (r_j^T \cdot r_j^*)$

        // $p_{j+1} \leftarrow r_{j+1} + \beta_j \cdot p_j$
        dscal( n, beta, p, 1 );
        daxpy( n, 1.0, r, 1, p, 1 );

        // $p_{j+1}^* \leftarrow r_{j+1}^* + \beta_j \cdot p_j^*$
        dscal( n, beta, ps, 1 );
        daxpy( n, 1.0, rs, 1, ps, 1 );

        // Check for convergence
        converged = ...;
    }

    oski_DestroyVecView( r_view ); oski_DestroyVecView( x_view );
    oski_DestroyVecView( y_view ); oski_DestroyVecView( ys_view );
    oski_DestroyVecView( p_view ); oski_DestroyVecView( ps_view );
    free( workspace );
}
```

This example does not explicitly tune, but if this solver is embedded as a subroutine call in some larger non-linear solver, one could imagine calling **oski_TuneMat** just after a call to this routine.

5 A Tuning Transformation Language

This section provides an overview, illustrated by example, of a dynamically typed, procedural object-oriented scripting language for describing how to convert an input matrix into a tuned matrix data structure. The syntax of the language, OSKI-Lua, is based on the scripting language Lua [26]. A call to **oski_GetMatTransforms** (Section 3.5 on page 24) returns a program in this language. Moreover, a user may write her own sequence of transformations as a program in this language, and then call **oski_ApplyMatTransforms** to execute the program, thereby creating a specific data structure.

The rest of this section presents a number of examples intended to give the reader a sense of what the transformation language could look like. A detailed specification is forthcoming.

5.1 Basic transformations

The simplest OSKI-Lua program is one which performs no transformations on the input matrix data structure:

```
# No tuning
return InputMat;
```

All OSKI-Lua programs have a predefined object named InputMat represents the input matrix in CSR or CSC format, and must **return** a matrix object. This program simply returns the input matrix as-is.

The input matrix is transformed by applying functions that create new matrix objects, usually in different data structures. These alternative ("tuned") matrix data structures are defined as types in the language, and instantiating an object of a particular type typically transforms an existing data structure into a new data structure. The following example converts the input matrix into a 4×2 block compressed sparse row (BCSR) format:

```
# Convert input matrix to $4 \times 2$ register blocked format
return BCSR:new( InputMat, 4, 2 );
```

Here, **BCSR** is a type and **new** is a method that takes a matrix object and block size as input, and returns a matrix object in the new format. Since instantiating objects in this way is quite common, we define a shorthand notation in which :**new** is omitted, *i.e.*,

```
return BCSR( InputMat, 4, 2 );
```

In our implementation, **BCSR**:**new**(A, r, c) has a native conversion routine for the case of A in either CSR or CSC format; any other format will be implicitly converted to one of these formats first.

5.2 A complex splitting example

The following example illustrates a more complex transformation. Let $A = A_1 + A_2 + A_3$ be a splitting of an input matrix A where A_1 is stored in 4×2 unaligned block compressed sparse row (UBCSR) format, A_2 is stored in 2×2 UBCSR, and A_3 is stored in CSR format:

```
1   # Let $A = A_1 + A_2 + A_3$ where
2   #     $A_1$ is in $4 \times 2$ UBCSR format
3   #     $A_2$ is in $2 \times 2$ UBCSR format
4   #     $A_3$ is in CSR format
5   T = VBR( InputMat );
6
7   # First, split $A = A_1 + A_{\mathrm{leftover}}$,
8   # where $A_{\mathrm{leftover}}$ is in CSR format
9   A1, A_leftover = T.extract_blocks(  4,  2  );
10
11  # Next, split $A_{\mathrm{leftover}} = A_2 + A_3$
12  T = VBR( A_leftover );
13  A2, A3 = T.extract_blocks(  2,  2  );
14
15  return A1 + A2 + A3;
```

(UBCSR essentially adds an additional set of row indices to BCSR format to allow a flexible alignment of block rows.)

The splitting in this example is based on first converting the input matrix to variable block row (VBR) format (line 5): the rows are partitioned into block rows of varying size, the columns into block columns, and only non-zero blocks are stored. The constructor **VBR** determines this partitioning automatically, though other optional arguments exist for controlling how the partitions are formed.

The **VBR** type has a method, **extract_blocks**, which greedily extracts blocks of the specified size, and returns two matrix objects: one in UBCSR containing all blocks, and the other in CSR to hold the leftover elements.[5] This method is first called (line 9) to extract 4×2 blocks, and repeated to extract 2×2 blocks from the leftovers (line 13).

[5]A naturally occuring 5×4 block will be extracted as two 4×2 blocks, with the leftover elements assigned to A_leftover.

The summation in the return statement (line 15) is a *symbolic* summation. The expression A1+A2+A3 implicitly evaluates to an object of type **SUM**. Indeed, this statement is equivalent to creating an object via

return **SUM**(A1, A2, A3);

A garbage collector automatically disposes of the temporary variables, T and A_leftover.

5.3 Example of reordering and splitting

A reordering transformation for creating dense blocks, based on approximating a solution to the Traveling Salesman Problem, is implemented as a function that returns row and column permutations and a reordered matrix in CSR. The following example computes a (symmetric) reordering $A = P^T \cdot \hat{A} \cdot P$, where the reordered matrix $\hat{A}$ is further split into a sum $A_1 + A_2$ where A_1 is stored in 2×2 BCSR, and A_2 is stored in CSR.

```
1  # Compute $\hat{A} = P \cdot A \cdot P^T$
2  A_hat, P = reorder_TSP( InputMat );
3
4  # Split: $\hat{A} = A_1 + A_2$, where $A_1$ is $2 \times 2$ BCSR and $A_2$ is CSR
5  A1, A2 = A_hat.extract_blocks( 2, 2 );
6
7  # Tuned result
8  A_tuned = transpose(P) * (A1 + A2) * P;
9  return A_tuned;
```

The function **reorder_TSP** (line 2) executes our implementation of Pinar and Heath's TSP-based reordering algorithm [36]. The first output variable, **A_hat**, stores the reordered matrix in CSR format. The second argument holds the row permutation. Since a column permutation is not assigned on output, **reorder_TSP** will apply the same permutation to both the rows and the columns.

The call to **extract_blocks** (line 5) uses a greedy algorithm to select 2×2 blocks from **A_hat**, and returns the result in BCSR format in A1. This **extract_blocks** method, a member of the CSR type, differs from the method of the same name used in the example of Section 5.2 on the preceding page, since that method was a member of the VBR (and not CSR) type.

The assignment to A_tuned (line 8) creates a matrix object which is symbolically equal to the right-hand side and which implicitly evaluates to creating an object via the statement:

A_tuned = **PERMFORM**(**transpose**(P), **SUM**(A1, A2), P);

5.4 Switch-to-dense example

We recently demonstrated the efficacy of a *switch-to-dense* optimization for sparse triangular solve [48]. If L is a lower triangular matrix, this transformation partitions L as follows:

$$L = \begin{pmatrix} L_1 & 0 \\ L_2 & L_D \end{pmatrix}$$

where L_1 is a sparse lower triangular matrix, L_2 is a sparse rectangular matrix, and L_D is a dense lower triangular matrix. In practice, L_D may account for as many as 90% of all of the non-zeros in L, and L_1 and L_2 may contain naturally occurring block structure [48].

OSKI-Lua as a special function, **split_s2d**, and type, **TRIPART**, for expressing a partition of this form.

```
    ASSERT( is_lower(InputMat) );
    L1, L2, LD = split_s2d( InputMat );
    return TRIPART( L1, L2, LD );
```

The ASSERT statement (line 1) can be used to verify that the input matrix satisfies certain properties. As written, the partition boundaries are determined by **split_s2d** automatically [48], but could also be specified explicitly:

```
    ASSERT( is_lower(InputMat) );
    switch_point = 25382;   # L_2 and L_D begin at row 25382
    L1, L2, LD = split_s2d( InputMat, switch_point );
    return TRIPART( L1, L2, LD );
```

6 Approaches that Complement Libraries

There are a number of complementary approaches to a library implementation. One is to implement a library using a language with generic programming constructs such as templates in C++ [34]. Both Blitz++ [45] and the Matrix Template Library (MTL) [41] have adopted this approach to building generic libraries in C++ that mimic dense BLAS functionality. The use of templates faciliates the generation of large numbers of library routines with relatively small amount of code, and flexibly handles issues of producing libraries that can handle different precisions. Sophisticated use of templates furthermore allows some limited optimization, such as unrolling. In some cases, loop-fusion like transformations have been implemented using templates [45]. However, this approach lacks an explicit mechanism for dealing with run-time search. Furthermore, the template mechanism for code generation can put enormous stress (in terms of memory and execution time) on the compiler.[6]

Another approach which extends the generic programming idea is compiler-based sparse code generation via restructuring compilers, pursued by Bik [8, 9, 10], Stodghill, *et al.* [42, 2, 32, 31], and Pugh and Shpeisman [37, 29]. These are clean, general approaches to code generation: the user expresses separately both the kernels (as dense code with random access to matrix elements) and a formal specification of a desired sparse data structure; a restructuring compiler combines the two descriptions to produce a sparse implementation. In addition, since any kernel can in principle be expressed, this overcomes a library approach in which all possible kernels must be pre-defined. Nevertheless, we view this technology as complementary to the overall library approach: while sparse compilers could be used to provide the underlying implementations of sparse primitives, they do not explicitly make use of matrix structural information available, in general, only at run-time.[7]

A third approach is to extend an existing library or system. There are a number of application-level libraries (*e.g.*, PETSc [6, 5], among others [21, 39, 38, 24]) and high-level application tools (*e.g.*, MATLAB [43, 22], Octave [18], approaches that apply compiler analyses and transformations to MATLAB code [3, 33]) that provide high-level sparse kernel support. Integration with these systems has a number of advantages, including the ability to hide data structure details and the tuning process from the user, and the large potential user base. However, our goal is to provide building blocks in the spirit of the BLAS with

[6]This concern is "practical" in nature and could be overcome through better compiler front-end technology. Another minor but related concern is the lack of consistency in how well aspects of the template mechanism are supported, making portability an issue.

[7]Technically, Bik's sparse compiler does use matrix non-zero structure information [10], but is restricted in the following two senses: (1) it assumes that the matrix is available at "compile-time," and (2) it supports a limited number of fixed data structures.

the steps and costs of tuning exposed. This model of development has been very successful with other numerical libraries, examples of which include the integration of ATLAS and FFTW tuning systems into the commercial MATLAB system. Thus, it should be possible to integrate the OSKI library into an existing system as well.

Acknowledgements

We thank Rajesh Nishtala, Rob Schreiber, Matt Knepley, Michael Heroux, and Viral Shah, for discussion on the key ideas behind and early drafts of this document.

References

[1] *Berkeley Benchmarking and OPtimization (BeBOP) Group home page*, 2004. `bebop.cs.berkeley.edu`.

[2] N. Ahmed, N. Mateev, K. Pingali, and P. Stodghill, *A framework for sparse matrix code synthesis from high-level specifications*, in Proceedings of Supercomputing 2000, Dallas, TX, November 2000.

[3] G. Almási and D. Padua, *MaJIC: Compiling MATLAB for speed and responsiveness*, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

[4] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., *Templates for the solution of algebraic eigenvalue problems: a practical guide*, SIAM, Philadelphia, PA, USA, 2000.

[5] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc User's Manual*, Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2002. www.mcs.anl.gov/petsc.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

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

[8] A. J. C. Bik, *Compiler Support for Sparse Matrix Codes*, PhD thesis, Leiden University, 1996.

[9] A. J. C. Bik, P. J. H. Birkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff, *The automatic generation of sparse primitives*, ACM Transactions on Mathematical Software, 24 (1998), pp. 190–225.

[10] A. J. C. Bik and H. A. G. Wijshoff, *Automatic nonzero structure analysis*, SIAM Journal on Computing, 28 (1999), pp. 1576–1587.

[11] J. Bilmes, K. Asanović, C. Chin, and J. Demmel, *Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology*, in Proceedings of the International Conference on Supercomputing, Vienna, Austria, July 1997, ACM SIGARC.

[12] S. L. Blackford, J. W. Demmel, J. Dongarra, I. S. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, *An updated set of basic linear algebra subprograms (BLAS)*, ACM Transactions on Mathematical Software, 28 (2002), pp. 135–151.

[13] D. A. Burgess and M. B. Giles, *Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines*, tech. rep., Numerical Analysis Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 1995.

[14] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the ACM National Conference, 1969.

[15] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997.

[16] I. S. Duff, M. A. Heroux, and R. Pozo, *An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum*, ACM Transactions on Mathematical Software, 28 (2002), pp. 239–267.

[17] I. S. Duff and C. Vömel, *Algorithm 818: A reference model implementation of the sparse BLAS in Fortran 95*, ACM Transactions on Mathematical Software, 28 (2002), pp. 268–283.

[18] J. W. Eaton, *Octave*, 2003. `www.octave.org`.

[19] S. Filippone and M. Colajanni, *PSBLAS: A library for parallel linear algebra computation on sparse matrices*, ACM Transactions on Mathematical Software, 26 (2000), pp. 527–550.

[20] M. Frigo and S. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Seattle, Washington, May 1998.

[21] A. George and J. W. H. Liu, *The design of a user interface for a sparse matrix package*, ACM Transactions on Mathematical Software, 5 (1979), pp. 139–162.

[22] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: Design and implementation*, SIAM Journal on Matrix Analysis and Applications, 13 (1992), pp. 333–356.

[23] G. Heber, R. Biswas, and G. R. Rao, *Self-avoiding walks over adaptive unstructured grids*, Concurrency: Practice and Experience, 12 (2000), pp. 85–109.

[24] B.-O. Heimsund, *JMP: A sparse matrix library in Java*, 2003. `http://www.mi.uib.no/~bjornoh/jmp`.

[25] C. Hsu, *Effects of block size on the block Lanczos algorithm*, June 2003. Senior thesis.

[26] R. Ierusalimschy, L. H. de Figeiredo, and W. Celes, *Lua 5.0 Reference Manual*, Tech. Rep. MCC-14/03, PUC-Rio, April 2003. `www.lua.org`.

[27] E.-J. Im and K. A. Yelick, *Optimizing sparse matrix vector multiplication on SMPs*, in Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, USA, March 1999.

[28] E.-J. Im, K. A. Yelick, and R. Vuduc, *SPARSITY: Framework for optimizing sparse matrix-vector multiply*, International Journal of High Performance Computing Applications, 18 (2004), pp. 135–158.

[29] J. Irwin, J.-M. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman, *Aspect-oriented programming of sparse matrix code*, in Proceedings of the International Scientific Computing in Object-Oriented Parallel Environments, Marina del Rey, CA, USA, December 1997.

[30] B. C. LEE, R. W. VUDUC, J. W. DEMMEL, K. A. YELICK, M. DELORIMIER, AND L. ZHONG, *Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply*, Tech. Rep. UCB/CSD-03-1297, University of California, Berkeley, Berkeley, CA, USA, November 2003.

[31] N. MATEEV, K. PINGALI, AND P. STODGHILL, *The Bernoulli Generic Matrix Library*, Tech. Rep. TR-2000-1808, Cornell University, 2000.

[32] N. MATEEV, K. PINGALI, P. STODGHILL, AND V. KOTLYAR, *Next-generation generic programming and its application to sparse matrix computations*, in International Conference on Supercomputing, 2000.

[33] V. MENON AND K. PINGALI, *A case for source-level transformations in MATLAB*, in Proceedings of the 2nd Conference on Domain-Specific Languages, Austin, TX, October 1999.

[34] D. R. MUSSER AND A. A. STEPANOV, *Algorithm-oriented generic libraries*, Software: Practice and Experience, 24 (1994), pp. 632–642.

[35] R. NISHTALA, R. VUDUC, J. DEMMEL, AND K. YELICK, *When cache blocking sparse matrix vector multiply works and why*, in Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing, Copenhagen, Denmark, June 2004. (to appear).

[36] A. PINAR AND M. HEATH, *Improving performance of sparse matrix-vector multiplication*, in Proceedings of Supercomputing, 1999.

[37] W. PUGH AND T. SHPEISMAN, *Generation of efficient code for sparse matrix computations*, in Proceedings of the 11th Workshop on Languages and Compilers for Parallel Computing, LNCS, August 1998.

[38] K. REMINGTON AND R. POZO, *NIST Sparse BLAS: User's Guide*, tech. rep., NIST, 1996. `gams.nist.gov/spblas`.

[39] Y. SAAD, *SPARSKIT: A basic toolkit for sparse matrix computations*, 1994. `www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`.

[40] ——, *Iterative methods for sparse linear systems*, SIAM, 2003.

[41] J. G. SIEK AND A. LUMSDAINE, *A rational approach to portable high performance: the Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (fast) library*, in Proceedings of ECOOP, Brussels, Belgium, 1998.

[42] P. STODGHILL, *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*, PhD thesis, Cornell University, August 1997.

[43] I. THE MATHWORKS, *Matlab*, 2003. `www.mathworks.com`.

[44] S. TOLEDO, *Improving memory-system performance of sparse matrix-vector multiplication*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, March 1997.

[45] T. VELDHUIZEN, *Arrays in Blitz++*, in Proceedings of ISCOPE, vol. 1505 of LNCS, Springer-Verlag, 1998.

[46] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, *Performance optimizations and bounds for sparse matrix-vector multiply*, in Proceedings of Supercomputing, Baltimore, MD, USA, November 2002.

[47] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick, *Memory hierarchy optimizations and bounds for sparse A^TAx*, in Proceedings of the ICCS Workshop on Parallel Linear Algebra, P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya, and Y. E. Gorbachev, eds., vol. LNCS 2660, Melbourne, Australia, June 2003, Springer, pp. 705–714.

[48] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick, *Automatic performance tuning and analysis of sparse triangular solve*, in ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries, New York, USA, June 2002.

[49] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*, PhD thesis, University of California, Berkeley, December 2003.

[50] R. C. Whaley, A. Petitet, and J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–25.

A Valid input matrix representations

The user creates a sparse matrix object in our interface from a pre-assembled *input matrix*. At present, we support the matrix representations listed below. Each representation defines a mathematical matrix A of size $m \times n$ whose element values we denote by $A(i, j)$ where $1 \leq i \leq m$ and $1 \leq j \leq n$.

- **Packed 3-array compressed sparse row using 0-based indices**: The user provides 3 arrays corresponding to A: Aptr, Aind, whose elements are non-negative integers, and Aval, whose elements are real or complex values. These arrays satisfy the following conditions:

  1. Aptr is of length at least Aptr[$m + 1$], Aptr[0] ≥ 0, and for all $0 \leq i < m$, Aptr[i] $\leq$ Aptr[$i + 1$].
  2. Aind is of length at least Aptr[m]. Each element of Aind lies between 0 and $n-1$ inclusive.
  3. Aval is of length at least Aptr[m].

  A matrix element $A(i, j)$ is computed from this representation as follows. Let K_{ij} be the set $\{k$: Aptr[$i - 1$] $\leq k <$ Aptr[i] and Aind[k] $= j - 1\}$. Then $A(i, j) = \sum_{k \in K}$ Aval[k]. (That is, repeated elements are summed.)

- **Packed 3-array compressed sparse row using 1-based indices**: The user provides 3 arrays, Aptr, Aind, and Aval corresponding to A. These arrays satisfy the following conditions:

  1. Aptr is of length at least Aptr[$m + 1$], Aptr[0] ≥ 1, and for all $0 \leq i < m$, Aptr[i] $\leq$ Aptr[$i + 1$].
  2. Aind is of length at least Aptr[m]. Each element of Aind lies between 1 and n inclusive.
  3. Aval is of length at least Aptr[m].

  A matrix element $A(i, j)$ is computed from this representation as follows. Let K_{ij} be the set $\{k$: Aptr[$i-1$] $\leq k <$ Aptr[i] and Aind[k] $= j\}$. Then $A(i, j) = \sum_{k \in K}$ Aval[k]. (Repeated elements are summed.)

- **Packed 3-array compressed sparse column using 0-based indices**: The user provides 3 arrays, Aptr, Aind, and Aval corresponding to A. These arrays satisfy the following conditions:

  1. Aptr is of length at least Aptr[$n + 1$], Aptr[0] ≥ 0, and for all $0 \leq j < n$, Aptr[j] $\leq$ Aptr[$j + 1$].
  2. Aind is of length at least Aptr[n]. Each element of Aind lies between 0 and $m-1$ inclusive.
  3. Aval is of length at least Aptr[n].

  A matrix element $A(i, j)$ is computed from this representation as follows. Let K_{ij} be the set $\{k$: Aptr[$j - 1$] $\leq k <$ Aptr[j] and Aind[k] $= i - 1\}$. Then $A(i, j) = \sum_{k \in K}$ Aval[k]. (Repeated elements are summed.)

- **Packed 3-array compressed sparse column using 1-based indices**: The user provides 3 arrays, Aptr, Aind, and Aval corresponding to A. These arrays satisfy the following conditions:

1. Aptr is of length at least Aptr[$n + 1$], Aptr[0] ≥ 1, and for all $0 \leq j < n$, Aptr[j] $\leq$ Aptr[$j + 1$].
2. Aind is of length at least Aptr[n]. Each element of Aind lies between 1 and m inclusive.
3. Aval is of length at least Aptr[n].

A matrix element $A(i, j)$ is computed from this representation as follows. Let K_{ij} be the set $\{k$: Aptr[$j - 1$] $\leq k <$ Aptr[j] and Aind[k] $= i\}$. Then $A(i, j) = \sum_{k \in K}$ Aval[k]. (Repeated elements are summed.)

B Bindings Reference

We define each routine in the interface using the formatting conventions used in the following example for a function to compute the factorial of a non-negative integer:

---

int
factorial (int **n**);

Given an integer $n \geq 0$, returns $n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$ if $n \geq 1$, or 1 if $n = 0$.

Parameters:
n [input] **n** ≥ 0
Non-negative integer of which to compute a factorial.

Actions and Returns:
An integer whose value equals n! if n is greater than 1, or 1 if n equals 0. The return value is undefined if n! exceeds the maximum positive integer of type int.

Error conditions and actions:
Aborts program if n is less than 0.

Example:
```
int n = 4;
int ans = factorial ( n );
printf( "%d! == %d\n", n, ans ); // Should print '4! == 24'
```

The specification indicates any argument preconditions (under "**Parameters**:"), return values and side effects ("**Actions and Returns**:"), possible error conditions and actions ("**Error conditions and actions**:"), and short usage examples ("**Example**:").

As discussed in Section 3.6 on page 24, the interface provides two error-handling mechanisms: return codes and error-handling functions. By convention, readers can assume that any routine returning integers (type int) will return negative values on errors. In addition, all routines call an error handler if one is available in a given context according to the process described in Section 3.6. For all routines, a violation of argument preconditions is always considered an error condition.

Many of the specifications refer to the mathematical matrix A defined by a given matrix object. We take this matrix to have dimensions $m \times n$, and with elements $A(i, j)$ referenced beginning at position $(1, 1)$, *i.e.*, $1 \leq i \leq m$ and $1 \leq j \leq n$. However, since we are presenting the C interface, note that all *array* indexing will be zero-based.

B.1 Matrix object creation and modification

---

oski_matrix_t
oski_CreateMatCSR(
 oski_index_t∗ **Aptr**, **oski_index_t**∗ **Aind**, **oski_value_t**∗ **Aval**,
 oski_index_t **num_rows**, **oski_index_t** **num_cols**,
 oski_copymode_t **mode**,
 int **k**, [**oski_inmatprop_t** **property_1**, . . ., **oski_inmatprop_t** **property_k**]);

Creates and returns a valid tunable matrix object from a compressed sparse row (CSR) representation.

Parameters:

num_rows × **num_cols** [input] — **num_rows** ≥ 0, **num_cols** ≥ 0
Dimensions of the input matrix.

Aptr, **Aind**, **Aval** [input] — **Aptr**, **Aind**, **Aval** ≠NULL
The input matrix pattern and values must correspond to a valid CSR representation, as defined in Appendix A on page 38.

mode [input] — See Table 2 on page 16.
Specifies the copy mode for the arrays **Aptr**, **Aind**, and **Aval**. See Section 3.2.1 on page 14 for a detailed explanation.

k [input] — **k** ≥0
The number of qualifying properties.

property_1, . . . **property_k** [input; optional] — See Table 3 on page 17.
The user may assert that the input matrix satisfies zero or more properties listed in Table 3 on page 17. Grouped properties are mutually exclusive, and specifying two or more properties from the same group generates an error (see below). The user must supply exactly**k** properties.

Actions and Returns:
A valid, tunable matrix object, or INVALID_MAT on error. Any kernel operations or tuning operations may be called using this object.

Error conditions and actions:
Possible error conditions include:

1. Any of the argument preconditions above are not satisfied [ERR_BAD_ARG].
2. More than 1 property from the same group are specified (see Table 3 on page 17) [ERR_INMATPROP_CONFLICT].
3. The input matrix arrays do not correspond to a valid CSR representation [ERR_NOT_CSR], or are incompatible with any of the asserted properties [ERR_FALSE_INMATPROP]. As an example of the latter error, if the user asserts that the matrix is symmetric but the number of rows is not equal to the number of columns, then an error is generated.

Example:
See Listing 1 on page 8.

---

oski_matrix_t
oski_CreateMatCSC(
    oski_index_t∗ **Aptr**, **oski_index_t**∗ **Aind**, **oski_value_t**∗ **Aval**,
    oski_index_t **num_rows**, **oski_index_t** **num_cols**,
    oski_copymode_t **mode**,
    int **k**, [**oski_inmatprop_t** **property_1**, . . ., **oski_inmatprop_t** **property_k**]);

Creates and returns a valid tunable matrix object from a compressed sparse column (CSC) representation.

Parameters:
num_rows × **num_cols** [input] — **num_rows** ≥ 0, **num_cols** ≥ 0
Dimensions of the input matrix.

Aptr, **Aind**, **Aval** [input] — **Aptr**, **Aind**, **Aval** ≠NULL
The input matrix pattern and values must correspond to a valid CSC representation, as defined in Appendix A on page 38.

mode [input] — See Table 2 on page 16.
Specifies the copy mode for the arrays **Aptr**, **Aind**, and **Aval**. See Section 3.2.1 on page 14 for a detailed explanation.

k [input] **k** ≥ 0
The number of qualifying properties.

property_1, . . . **property_k** [input; optional] See Table 3 on page 17.
The user may assert that the input matrix satisfies zero or more properties listed in Table 3 on page 17. Grouped properties are mutually exclusive, and specifying two or more properties from the same group generates an error (see below).

Actions and Returns:
A valid, tunable matrix object, or INVALID_MAT on error. Any kernel operations or tuning operations may be called using this object.

Error conditions and actions:
Possible error conditions include:
1. Any of the argument preconditions above are not satisfied [ERR_BAD_ARG].
2. More than 1 property from the same group are specified (see Table 3 on page 17) [ERR_INMATPROP_CONFLICT].
3. The input matrix arrays do not correspond to a valid CSC representation [ERR_NOT_CSC], or are incompatible with any of the asserted properties [ERR_FALSE_INMATPROP].

---

oski_value_t
oski_GetMatEntry(const **oski_matrix_t** **A_tunable**, **oski_index_t** **row**, **oski_index_t** **col**);

Returns the value of a matrix element.

Parameters:
A_tunable [input] **A_tunable**is valid.
The object representing some $m \times n$ matrix A.

row, **col** [input] $1 \leq$ **row** $\leq m$, $1 \leq$ **col** $\leq n$
Specifies the element whose value is to be returned. The precondition above must be satisfied. Note that matrix entries are referenced using 1-based indices, regardless of the convention used when the matrix was created.

Actions and Returns:
If **row** and **col** are valid, then this routine returns the value of the element A(**row**,**col**). Otherwise, it returns NaN_VALUE.

Error conditions and actions:
Possible error conditions include:
1. Invalid matrix [ERR_BAD_MAT].
2. Position **row**, **col** is out-of-range [ERR_BAD_ENTRY].

Example:
```
// Let A be the matrix shown in Listing 1 on page 8, and stored in A_tunable.
// The following should prints "A(2,2) = 1", "A(2,3) = 0", and "A(3,1) = .5"
printf( "A(2,2) = %f\n", oski_GetMatEntry(A_tunable, 2, 2) );
printf( "A(2,3) = %f\n", oski_GetMatEntry(A_tunable, 2, 3) );
printf( "A(3,1) = %f\n", oski_GetMatEntry(A_tunable, 3, 1) );
```

---

int
oski_SetMatEntry(**oski_matrix_t** **A_tunable**, **oski_index_t** **row**, **oski_index_t** **col**,
  oski_value_t **val**);

Changes the value of the specified matrix element.

Parameters:

A_tunable [input/output] — A_tunable is valid

The object representing some $m \times n$ matrix A.

row, **col** [input] — $1 \le$ **row** $\le m$, $1 \le$ **col** $\le n$

Specifies the element whose value is to be modified. This element **must** have had an associated element stored explicitly in the input matrix when **A_tunable** was created.

If the user asserted that her input matrix was symmetric or Hermitian when the matrix was created, these properties are preserved with this change in value. In contrast, asserting a *tuning hint* to say the matrix is structurally symmetric does not cause this routine to insert both $A(i,j)$ and $A(j,i)$.

Actions and Returns:

Returns 0 and sets $A($**row**,**col**$) \leftarrow$**val**. If the matrix was created as either symmetric or Hermitian (including the semantic properties, MAT_SYMM_FULL and MAT_HERM_FULL), this routine logically sets $A($**col**,**row**$)$ to be **val** also. On error, **A_tunable** remains unchanged and an error code is returned.

NOTE: When **A_tunable** is tuned, the tuned data structure may store additional explicit zeros to improve performance. The user should avoid changing entries that were not explicitly stored when **A_tunable** was created.

Error conditions and actions:

Possible error conditions include:

1. Invalid matrix [ERR_BAD_MAT].
2. The position (**row**,**col**) is out-of-range [ERR_BAD_ENTRY].
3. The position (**row**,**col**) was not explicitly stored when **A_tunable** was created (*i.e.*, the specified entry should always be logically zero) [ERR_ZERO_ENTRY]. This condition cannot always be enforced (*e.g.*, if tuning has replaced the data structure and freed the original), so this error will not always be generated.
4. Changing (**row**,**col**) would violate one of the asserted semantic properties when **A_tunable** was created [ERR_INMATPROP_CONFLICT]. For instance, suppose $A(i,j)$ is in the upper triangle of a matrix in which MAT_TRI_LOWER was asserted is an error condition; or, suppose the caller asks to change a diagonal element to a non-unit value when MAT_UNIT_DIAG_IMPLICIT was asserted.

Example:

// First, create $A = \begin{pmatrix} 1 & -2 & .5 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix}$, a sparse symmetric matrix with a unit diagonal.

```
int Aptr[] = {1, 3, 3, 3}, Aind[] = {1, 2};   // Uses 1-based indices!
double Aval[] = {−2, 0.5};

oski_matrix_t A_tunable = oski_CreateMatCSR( Aptr, Aind, Aval, 3, 3, SHARE_INPUTMAT,
    2, MAT_SYMM_UPPER, MAT_UNIT_DIAG_IMPLICIT );

printf( "A(1,3) = %f\n", oski_GetMatEntry(A_tunable, 1, 3) ); // prints "A(1,3) = 0.5"
printf( "A(3,1) = %f\n", oski_GetMatEntry(A_tunable, 3, 1) ); // prints "A(3,1) = 0.5"

// Change A(3,1) and A(1,3) to -.5.
oski_SetMatEntry( A_tunable, 3, 1, −.5 );

printf( "A(1,3) = %f\n", oski_GetMatEntry(A_tunable, 1, 3) ); // prints "A(1,3) = -0.5"
printf( "A(3,1) = %f\n", oski_GetMatEntry(A_tunable, 3, 1) ); // prints "A(3,1) = -0.5"
```

```
int
oski_GetMatClique( const oski_matrix_t A_tunable,
        const oski_index_t* rows, oski_index_t num_rows,
        const oski_index_t* cols, oski_index_t num_cols,
        oski_vecview_t vals );
```

Returns a block of values, defined by a clique, from a matrix.

Parameters:

A_tunable [input] — **A_tunable** is valid

The object representing some $m \times n$ matrix A.

num_rows, **num_cols** [input] — **num_rows** ≥ 1, **num_cols** ≥ 1

Dimensions of the block of values.

rows, **cols** [input] — **rows** $\neq$NULL, **cols** $\neq$NULL

Indices defining the block of values. The array **rows** must be of length at least **num_rows**, and **cols** must be of length at least **num_cols**. The entries of **rows** and **cols** must satisfy

- $1 \leq$**rows**$[i] \leq m$ for all $0 \leq i <$ **num_rows**, and
- $1 \leq$**cols**$[j] \leq n$ for all $0 \leq j <$ **num_cols**.

vals [output] — **vals** is valid.

The object **vals** is a multivector view (see Section 3.2.3 on page 16) of a logical two-dimensional array to be used to store the block of values. We use a view here to allow the user to specify row or column major storage and the leading dimension of the array.

Actions and Returns:

Let X be the **num_rows**$\times$**num_cols** matrix corresponding to **vals**. If **rows** and **cols** are valid (as discussed above), then this routine sets $X(r,c) \leftarrow A(i,j)$, where $i =$**rows**$[r-1]$ and $j =$**cols**$[c-1]$, for all $1 \leq r \leq$**num_rows** and $1 \leq c \leq$**num_cols**, and returns 0. Otherwise, this routine returns an error code and leaves X unchanged.

Error conditions and actions:

Possible errors conditions include:

1. Invalid matrix [ERR_BAD_MAT].
2. An invalid row, col was given [ERR_BAD_ENTRY].

Example:

```
// Let A be the matrix shown in Listing 1 on page 8, and stored in A_tunable.
int rows[] = { 1, 3 };
int cols[] = { 1, 3 };
double vals[] = { −1, −1, −1, −1 };

oski_vecview_t vals_view = oski_CreateMultiVecView( vals, 2, 2, LAYOUT_ROWMAJ, 2 );

oski_GetMatClique( A_tunable, rows, 2, cols, 2, vals_view );

printf( "A(1,1) == %f\n", vals[0] ); // prints "A(1,1) == 1"
printf( "A(1,3) == %f\n", vals[1] ); // prints "A(1,3) == 0"
printf( "A(3,1) == %f\n", vals[2] ); // prints "A(3,1) == 0.5"
printf( "A(3,3) == %f\n", vals[3] ); // prints "A(3,3) == 1"
```

```
int
oski_SetMatClique( oski_matrix_t A_tunable,
        const oski_index_t∗ rows, oski_index_t num_rows,
        const oski_index_t∗ cols, oski_index_t num_cols,
        const oski_vecview_t vals );
```

Changes a block of values, defined by a clique, in a matrix.

Parameters:

A_tunable [output] — A_tunable is valid

The object representing some $m \times n$ matrix A.

num_rows, **num_cols** [input] **num_rows** ≥ 1, **num_cols** ≥ 1
Dimensions of the block of values.

rows, **cols** [input] **rows** $\neq$NULL, **cols** $\neq$NULL
Indices defining the block of values. The array **rows** must be of length at least **num_rows**, and **cols** must be of length at least **num_cols**. The entries of **rows** and **cols** must satisfy

$1 \leq$**rows**$[i] \leq m$ for all $0 \leq i <$ **num_rows**, and

$1 \leq$**cols**$[j] \leq n$ for all $0 \leq j <$ **num_cols**.

vals [input] **vals** is valid.
The object **vals** is a multivector view (see Section 3.2.3 on page 16) of a logical two-dimensional array to be used to store the block of values. We use a view here to allow the user to specify row or column major storage and the leading dimension of the array.

Actions and Returns:
Let X be the **num_rows**$\times$**num_cols** matrix corresponding to **vals**. If **rows** and **cols** are valid (as discussed above), then this routine sets $A(i,j) \leftarrow X(r,c)$, where $i =$**rows**$[r-1]$ and $j =$**cols**$[c-1]$, for all $1 \leq r \leq$**num_rows** and $1 \leq c \leq$**num_cols**, and returns 0. Otherwise, this routine returns an error code and leaves X unchanged.

If the matrix was created as either symmetric or Hermitian (including the semantic properties, MAT_SYMM_FULL and MAT_HERM_FULL), this routine logically sets $A(i,j)$ and $A(j,i)$. If both (i,j) and (j,i) are explicitly specified by the clique, the behavior is undefined if the corresponding values in **vals** are inconsistent.

If an entry $A(i,j)$ is specified by the clique and appears multiple times within the clique with inconsistent values in **vals**, the behavior is undefined.

NOTE: When **A_tunable** is tuned, the tuned data structure may store additional explicit zeros to improve performance. The user should avoid changing entries that were not explicitly stored when **A_tunable** was created.

Error conditions and actions:
Possible error conditions include:

1. Invalid matrix [ERR_BAD_MAT].
2. The position (row,col) is out-of-range [ERR_BAD_ENTRY].
3. The position (row,col) was not explicitly stored when **A_tunable** was created (*i.e.*, the specified entry should always be logically zero) [ERR_ZERO_ENTRY]. This condition cannot always be enforced (*e.g.*, if tuning has replaced the data structure and freed the original), so this error will not always be generated.
4. Changing (row,col) would violate one of the asserted semantic properties when **A_tunable** was created [ERR_INMATPROP_CONFLICT]. For instance, suppose $A(i,j)$ is in the upper triangle of a matrix in which MAT_TRI_LOWER was asserted is an error condition; or, suppose the caller asks to change a diagonal element to a non-unit value when MAT_UNIT_DIAG_IMPLICIT was asserted.

Example:

// First, create $A = \begin{pmatrix} 1 & -2 & .5 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix}$, a sparse symmetric matrix with a unit diagonal.

```
int  Aptr[] = {1,  3,  3,  3},  Aind[] = {1,  2};    // Uses 1-based indices!
double Aval[] = {−2, 0.5};

oski_matrix_t A_tunable = oski_CreateMatCSR( Aptr, Aind, Aval, 3, 3, SHARE_INPUTMAT,
    2,  MAT_SYMM_UPPER, MAT_UNIT_DIAG_IMPLICIT );
```

// Clique of values to change, using 1-based indices to match matrix.

// The new values are $\begin{pmatrix} 1 & .125 \\ .125 & 1 \end{pmatrix}$.

```
int  rows[] = { 1,  2 };
int  cols[]  = { 1,  2 };
double vals[]  = { −1, −1, −1, −1 };   // in row major order
```

double new_vals[] = { 1, .125, .125, 1 }; // in row major order

oski_vecview_t **vals_view** = **oski_CreateMultiVecView**(**vals**, 2, 2, LAYOUT_ROWMAJ, 2);
oski_vecview_t **new_vals_view** = **oski_CreateMultiVecView**(new_vals, 2, 2, LAYOUT_ROWMAJ, 2);

// Retrieve the submatrix of values, $\begin{pmatrix} 1 & -2 \\ -2 & 1 \end{pmatrix}$.
oski_GetMatClique(**A_tunable**, **rows**, 2, **cols**, 2, **vals_view**);
printf("A(1,1) == %f\n", **vals**[0]); // prints "A(1,1) == 1"
printf("A(1,2) == %f\n", **vals**[1]); // prints "A(1,2) == -2"
printf("A(2,1) == %f\n", **vals**[2]); // prints "A(2,1) == -2"
printf("A(2,2) == %f\n", **vals**[3]); // prints "A(2,2) == 1"

// Change the above values to $\begin{pmatrix} 1 & .125 \\ .125 & 1 \end{pmatrix}$
oski_SetMatClique(**A_tunable**, **rows**, 2, **cols**, 2, **new_vals_view**);
oski_GetMatClique(**A_tunable**, **rows**, 2, **cols**, 2, **vals_view**);
printf("A(1,1) == %f\n", **vals**[0]); // prints "A(1,1) == 1"
printf("A(1,2) == %f\n", **vals**[1]); // prints "A(1,2) == 0.125"
printf("A(2,1) == %f\n", **vals**[2]); // prints "A(2,1) == 0.125"
printf("A(2,2) == %f\n", **vals**[3]); // prints "A(2,2) == 1"

---

int
oski_GetMatDiagValues(const **oski_matrix_t** **A_tunable**, **oski_index_t** **diag_num**,
  oski_vecview_t **diag_vals**);

Extract the diagonal d from A, *i.e.*, all entries $A(i, j)$ such that $j - i = d$.

Parameters:
A_tunable [input] — **A_tunable** is valid.
The $m \times n$ matrix A from which to extract diagonal entries.

diag_num [input] — 1−m ≤**diag_num** ≤n−1
Number d of the diagonal to extract.

diag_vals [output] — **diag_vals** is a valid view.
Let X be the $r \times s$ (multi)vector object into which to store the diagonal values, such that $s \geq 1$ and r is at least the length of the diagonal, *i.e.*, $r \geq \min\{\max\{m, n\} - d, \min\{m, n\}\}$.

Actions and Returns:
For all $j - i = d$, stores $A(i, j)$ in $X(k, 1)$, where $k = \min\{i, j\}$, and returns 0. On error, returns an error code.

Error conditions and actions:
Possible error conditions include:
1. Providing an invalid matrix [ERR_BAD_MAT].
2. Providing an invalid vector view, or a vector view with invalid dimensions [ERR_BAD_VECVIEW].
3. Specifying an invalid diagonal [ERR_BAD_ARG].

Example:

// First, create $A = \begin{pmatrix} 1 & -2 & .5 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix}$, a sparse symmetric matrix with a unit diagonal.
int Aptr[] = {1, 3, 3, 3}, Aind[] = {1, 2}; // Uses 1-based indices!
double Aval[] = {−2, 0.5};
double **diag_vals**[] = { 0, 0, 0 };

```
oski_vecview_t diag_vals_view = oski_CreateVecView( diag_vals, 3, STRIDE_UNIT );

oski_matrix_t A_tunable = oski_CreateMatCSR( Aptr, Aind, Aval, 3, 3, SHARE_INPUTMAT,
    2, MAT_SYMM_UPPER, MAT_UNIT_DIAG_IMPLICIT );

// Prints "Main diagonal = [1, 1, 1]"
oski_GetMatDiagValues( A_tunable, 0, diag_vals_view );
printf( "Main diagonal = [%f, %f, %f]\n", diag_vals[0], diag_vals[1], diag_vals[2] );

// Prints "First superdiagonal = [-2, 0]"
oski_GetMatDiagValues( A_tunable, 1, diag_vals_view );
printf( "First superdiagonal = [%f, %f]\n", diag_vals[0], diag_vals[1] );

// Prints "Second subdiagonal = [0.5]"
oski_GetMatDiagValues( A_tunable, −2, diag_vals_view );
printf( "Second subdiagonal = [%f]\n", diag_vals[0] );
```

---

int
oski_SetMatDiagValues(**oski_matrix_t** **A_tunable**, **oski_index_t** **diag_num**,
    const **oski_vecview_t** **diag_vals**);

Sets the values along diagonal d of A, *i.e.*, all entries $A(i, j)$ such that $j - i = d$.

Parameters:

A_tunable [input/output] **A_tunable** is valid.
The $m \times n$ matrix A in which to change diagonal entries.

diag_num [input] 1−m ≤**diag_num** ≤n−1
Number d of the diagonal to change.

diag_vals [output] **diag_vals** is a valid view.
Let X be the $r \times s$ (multi)vector object into which to store the diagonal values, such that $s \geq 1$ and r is at least the length of the diagonal, *i.e.*, $r \geq \min\{\max\{m, n\} - d, \min\{m, n\}\}$.

Actions and Returns:
For all $j - i = d$ such that $A(i, j)$ was an explicitly stored entry when **A_tunable** was created, sets $A(i, j) \leftarrow X(k, 1)$, where $k = \min\{i, j\}$, and returns 0. On error, returns an error code and leaves **A_tunable** unchanged.

If the matrix was created as either symmetric or Hermitian (including the semantic properties, MAT_SYMM_FULL and MAT_HERM_FULL), this routine also (logically) changes the corresponding symmetric diagonal −**diag_num**.

NOTE: When **A_tunable** is tuned, the tuned data structure may store additional explicit zeros to improve performance. The user should avoid changing entries that were not explicitly stored when **A_tunable** was created. If the user attempts to change such an entry by specifying a non-zero value in a corresponding entry of **diag_vals**, the value may or may not be changed.

Error conditions and actions:
Possible error conditions include:

1. Providing an invalid matrix [ERR_BAD_MAT].
2. Providing an invalid vector view, or a vector view with invalid dimensions [ERR_BAD_-VECVIEW].
3. Specifying an invalid diagonal [ERR_BAD_ENTRY].
4. Specifying the main diagonal when **A_tunable** was created with MAT_UNIT_DIAG_IMPLI-CIT.

Example:

// First, create $A = \begin{pmatrix} 1 & -2 & .5 \\ -2 & 1 & .25 \\ .5 & 0 & 1 \end{pmatrix}$, a sparse symmetric matrix with a unit diagonal.
int Aptr[] = {1, 3, 4, 4}, Aind[] = {1, 2, 3}; // Uses 1-based indices!
double Aval[] = {−2, 0.5, 0.25};
double **diag_vals**[] = { 0, 0, 0 };
oski_vecview_t diag_vals_view = **oski_CreateVecView**(**diag_vals**, 3, STRIDE_UNIT);

oski_matrix_t **A_tunable** = oski_CreateMat_CSR(Aptr, Aind, Aval, 3, 3, SHARE_INPUTMAT,
    2, MAT_SYMM_UPPER, MAT_UNIT_DIAG_IMPLICIT);

// Prints "First superdiagonal = [-2, 0.25]"
oski_GetMatDiagValues(**A_tunable**, 1, diag_vals_view);
printf(`"First superdiagonal = [%f, %f]\n"`, **diag_vals**[0], **diag_vals**[1]);

// Change first superdiagonal to be [-1, -2]
diag_vals[0] = −1;
diag_vals[1] = −2;
oski_SetMatDiagValues(**A_tunable**, 1, diag_vals_view);

// Prints "First superdiagonal = [-1, -2]"
diag_vals[0] = 0;
diag_vals[1] = 0;
oski_GetMatDiagValues(**A_tunable**, 1, diag_vals_view);
printf(`"First superdiagonal = [%f, %f]\n"`, **diag_vals**[0], **diag_vals**[1]);

---

oski_matrix_t
oski_CopyMat(const **oski_matrix_t** **A_tunable**);

Creates a copy of a matrix object.

Parameters:
A_tunable [input] A_tunable is valid
The object representing some $m \times n$ matrix A.

Actions and Returns:
Returns a new matrix object, or INVALID_MAT on error. The new matrix object is equivalent to the matrix object the user would obtain if she performed the following steps:
1. Re-execute the original call to **oski_CreateMatCSR**/**oski_CreateMatCSC** to create a new, untuned matrix object, **A_copy**, in the copy mode COPY_INPUTMAT. Thus, **A_copy** may exist independently of **A_tunable** and of any data upon which **A_tunable** might depend.
2. Get the tuning transformations that have been applied to **A_tunable** by the time of this call. Equivalently, execute oski_GetMatTransformations(**A_tunable**) and store the result.
3. Apply these transformations to **A_copy**.

Error conditions and actions:
Possible error conditions include an invalid source matrix object [ERR_BAD_MAT] or an out-of-memory condition while creating the clone [ERR_OUT_OF_MEMORY].

Example:
// Let A be the matrix shown in Listing 1 on page 8, and stored in A_tunable
// assuming zero-based indices.
int rows[] = { 0, 2 };
int cols [] = { 0, 2 };
double vals[] = { −1, −1, −1, −1 };

oski_vecview_t **vals_view** = **oski_CreateMultiVecView**(vals, 2, 2, LAYOUT_ROWMAJ, 2);

```
oski_matrix_t A_copy;

// For testing purposes, record and print a 2x2 clique of values.
oski_GetMatClique( A_tunable, rows, 2, cols, 2, vals_view );
printf( "A(1,1) == %f\n", vals[0] ); // prints "A(1,1) == 1"
printf( "A(1,3) == %f\n", vals[1] ); // prints "A(1,3) == 0"
printf( "A(3,1) == %f\n", vals[2] ); // prints "A(3,1) == 0.5"
printf( "A(3,3) == %f\n", vals[3] ); // prints "A(3,3) == 1"

// Create a clone
A_copy = oski_CopyMat( A_tunable );

// The clone is independent of the original, so we may delete the original.
oski_DestroyMat( A_tunable );

// Clear temporary clique value storage
memset( vals, 0, sizeof(double) * 4 );   // clear vals array
printf( "vals[0] == %f\n", vals[0] ); // prints "vals[0] == 0"
printf( "vals[1] == %f\n", vals[1] ); // prints "vals[1] == 0"
printf( "vals[2] == %f\n", vals[2] ); // prints "vals[2] == 0"
printf( "vals[3] == %f\n", vals[3] ); // prints "vals[3] == 0"

// Verify that the correct values were copied
oski_GetMatClique( A_copy, rows, 2, cols, 2, vals_view );
printf( "A(1,1) == %f\n", vals[0] ); // prints "A(1,1) == 1"
printf( "A(1,3) == %f\n", vals[1] ); // prints "A(1,3) == 0"
printf( "A(3,1) == %f\n", vals[2] ); // prints "A(3,1) == 0.5"
printf( "A(3,3) == %f\n", vals[3] ); // prints "A(3,3) == 1"
```

---

int
oski_DestroyMat(oski_matrix_t A_tunable);

Frees object memory associated with a given matrix object. The object is no longer usable.

Parameters:
A_tunable [input/output] A_tunable is valid
The object representing some $m \times n$ matrix A.

Actions and Returns:
Returns 0 if the object memory was fully successfully freed, or an error code on error.

Error conditions and actions:
Regardless of the return value, **A_tunable** should not be used after this call. Possible error conditions include an invalid matrix object [ERR_BAD_MAT].

Example:
See Listing 1 on page 8.

B.2 Vector object creation

---

oski_vecview_t
oski_CreateVecView(oski_value_t* x, oski_index_t length,
oski_index_t inc);

Creates a valid view on a single dense column vector x.

Parameters:
length [input] **length** ≥ 0
Number of vector elements.

inc [input] **inc** > 0
Stride, or distance in the user's dense array, between logically consecutive elements of x. Specifying STRIDE_UNIT is the same as setting **inc** = 1.

x [input] **x** $\neq$NULL
A pointer to the user's dense array representation of the vector x. Element x_i of the logical vector x, for all 1 $\leq i \leq$**length**, lies at position **x**[$(i-1)*$**inc**].

Actions and Returns:
Returns a valid vector view object for x, or INVALID_VEC on error.

Error conditions and actions:
An error occurs if any of the argument preconditions are not satisfied [ERR_BAD_ARG].

Example:
See Listing 1 on page 8.

---

oski_vecview_t
oski_CreateMultiVecView(oski_value_t* **X**,
oski_index_t **length**, **oski_index_t** **num_vecs**,
oski_storage_t **orient**, **oski_index_t** **lead_dim**);

Creates a view on k dense column vectors $X = (x_1 \cdots x_k)$, stored as a submatrix in the user's data.

Parameters:
length [input] **length** ≥ 0
Number of elements in each column vector.

num_vecs [input] **num_vecs** ≥ 0
The number of column vectors, *i.e.*, k as shown above.

orient [input]
Specifies whether the multivector is stored in row-major (LAYOUT_ROWMAJ) or column-major (LAYOUT_COLMAJ) order.

lead_dim [input] **lead_dim** ≥ 0
This parameter specifies the *leading dimension*, as specified in the BLAS standard. The leading dimension is the distance in **X** between the first element of each row vector, and must be at least **num_vecs**, if **orient** == LAYOUT_ROWMAJ. If instead **orient** == LAYOUT_COLMAJ, then the leading dimension is the distance in **X** between the first element of each column vector, and must be at least **length**.

X [input] **X** $\neq$NULL
Pointer to the user's dense array representation of X. For each 1 $\leq$ i $\leq$**length** and 1 $\leq$j $\leq$**num_vecs**, element x_{ij} (the i^{th} element of the j^{th} column, is stored at one of the following positions:

1. If **orient** == LAYOUT_ROWMAJ, then x_{ij} is stored at element **X**[$i*$**lead_dim** + j].
2. If **orient** == LAYOUT_COLMAJ, then x_{ij} is stored at element **X**[i + $j*$**lead_dim**].

Actions and Returns:
Returns a valid multivector view on the data stored in **X**, or INVALID_VEC on error.

Error conditions and actions:
Returns INVALID_VEC and calls the global error handler on an error. Possible error conditions include:
1. Any of the above argument preconditions are not satisfied [ERR_BAD_ARG].
2. The leading dimension is invalid for the specified storage orientation [ERR_BAD_LEAD-DIM].

Example:
// Let A be the matrix shown in Listing 1 on page 8, and stored in A_tunable,
// assuming zero-based indices.
// Let $Y = \begin{pmatrix} y_1 & y_2 \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{pmatrix}$ initially, and let $X = \begin{pmatrix} x_1 & x_2 \end{pmatrix} = \begin{pmatrix} .25 & -.25 \\ .45 & -.45 \\ .65 & -.65 \end{pmatrix}$
// The following example computes $Y \leftarrow Y + A \cdot X$.

double Y[] = { 1, −1, 1, −1, 1, −1 }; // in row-major order
oski_vecview_t **Y_view** = **oski_CreateMultiVecView**(Y, 3, 2, LAYOUT_ROWMAJ, 2);

double **X**[] = { .25, .45, .65, −.25, −.45, −.65 }; // in column-major order
oski_vecview_t **X_view** = **oski_CreateMultiVecView**(**X**, 3, 2, LAYOUT_COLMAJ, 3);

oski_MatMult(**A_tunable**, OP_NORMAL, 1, **X_view**, 1, **Y_view**);

// Views no longer needed.
oski_DestroyVecView(**X_view**);
oski_DestroyVecView(**Y_view**);

// Print result. Should be:
// "y1 = [1.25 ; 0.95 ; 1.775];"
// "y2 = [-1.25 ; -0.95 ; -1.775];"
printf(`"y1 = [ %f ; %f ; %f ];\n"`, Y[0], Y[2], Y[4]);
printf(`"y2 = [ %f ; %f ; %f ];\n"`, Y[1], Y[3], Y[5]);

---

int
oski_DestroyVecView(**oski_vecview_t** **x_view**);

Destroy a vector view.

Parameters:
x_view [input/output] **x_view** is valid
A vector view object to destroy. No action is taken if **x_view** is one of the predefined symbolic vectors, such as INVALID_VEC, SYMBOLIC_VEC, or SYMBOLIC_MULTIVEC.

Actions and Returns:
Returns 0 if the object memory (excluding the data on which this object views) was successfully freed, or an error code otherwise.

Error conditions and actions:
Regardless of the return value, **x_view** should not be returned after this call (unless **x_view** is equal to one of the predefined vector constants). The global error handler is called on error. Possible error conditions include providing an invalid vector [ERR_BAD_VECVIEW].

Example:
See Listing 1 on page 8, and the example for the routine **oski_CreateMultiVecView**.

---

oski_vecview_t
oski_CopyVecView(const **oski_vecview_t** **x_view**);

Creates a copy of the given (multi)vector view.

Parameters:
x_view [input] — **x_view** is valid.
A vector view object to clone.

Actions and Returns:
Returns another view object that views the same data as the source view object. If **x_view** is one of the symbolic vector constants (*e.g.*, INVALID_VEC, SYMBOLIC_VEC, SYMBOLIC_MULTIVEC), then that same constant is returned and no new object is created. On error, returns INVALID_VEC.

Error conditions and actions:
Returns INVALID_VEC on error, and calls the global error handler. Error conditions include specifying an invalid vector view object [ERR_BAD_VECVIEW], or an out-of-memory condition [ERR_OUT_OF_MEMORY].

Example:
// Let A, x, and y be as specified in Listing 1 on page 8 and stored in
// A_tunable, x_view, and y_view, respectively.

// Make a copy of the original view on x
oski_vecview_t **x_copy_view** = **oski_CopyVecView**(**x_view**);

// Dispose of original view
oski_DestroyVecView(**x_view**);

// Multiply with the copy
oski_MatMult(**A_tunable**, OP_NORMAL, -1, **x_copy_view**, 1, **y_view**);

// Finished with all objects
oski_DestroyMat(**A_tunable**);
oski_DestroyVecView(**x_copy_view**);
oski_DestroyVecView(**y_view**);

// Print result, y. Should be "[.75 ; 1.05 ; .225]"
printf(`"Answer: y = [ %f ; %f ; %f ]\n"`, y[0], y[1], y[2]);

B.3 Kernels

int
oski_MatMult(const **oski_matrix_t** **A_tunable**, **oski_matop_t** **opA**,
    oski_value_t **alpha**, const **oski_vecview_t** **x_view**,
    oski_value_t **beta**, **oski_vecview_t** **y_view**);

Computes $y \leftarrow \alpha \cdot \mathrm{op}(A) \cdot x + \beta \cdot y$, where $\mathrm{op}(A) \in \{A, A^T, A^H\}$.

Parameters:
A_tunable [input] — **A_tunable** is valid.
An object for a matrix A.

opA [input] — See Table 6 on page 19.
Specifies $\mathrm{op}(A)$.

alpha, **beta** [input]
Scalar constants α, β, respectively.

x_view [input] — **x_view** is valid.
View object for a (multi)vector x.

y_view [input/output] — **y_view** is valid.
View object for a (multi)vector y.

Actions and Returns:
Computes $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$ and returns 0 only if the dimensions of $\text{op}(A)$, x, and y are compatible. If the dimensions are compatible but any dimension is 0, this routine returns 0 but **y_view** is left unchanged. Otherwise, returns an error code and leaves **y_view** unchanged.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW], or incompatible input/output operand dimensions [ERR_DIM_MISMATCH].

Example:
See Listing 1 on page 8.

---

int
oski_MatTrisolve(const **oski_matrix_t** **T_tunable**, **oski_matop_t** **opT**,
    oski_value_t **alpha**, **oski_vecview_t** **x_view**);

Computes $x \leftarrow \alpha \cdot \text{op}(T)^{-1} \cdot x$, where T is a triangular matrix.

Parameters:
T_tunable [input] — **T_tunable** is valid, square, and triangular.
Matrix object for an $n \times n$ upper or lower triangular matrix T.

opT [input] — See Table 6 on page 19.
Specifies $\text{op}(T)$.

alpha [input]
Scalar constant α.

x_view [input/output] — **x_view** is valid.
View object for a (multi)vector x.

Actions and Returns:
If $\text{op}(T)$ and x have compatible dimensions, computes $x \leftarrow \alpha \cdot \text{op}(T)^{-1} \cdot x$ and returns 0. Otherwise, returns an error code.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

Example:
// Let A_tunable be object corresponding to the the sparse lower triangular matrix A
// shown in Listing 1 on page 8. The following example solves $A \cdot x = b$, where
// $b^T = \begin{pmatrix} .1 & 0 & .35 \end{pmatrix}$.

double **x**[] = { .1, 0, .35 };
oski_vecview_t **x_view** = **oski_CreateVecView**(**x**, 3, STRIDE_UNIT);

oski_MatTrisolve(**A_tunable**, OP_NORMAL, 1.0, **x_view**);

// Should print the solution, "x == [0.1 ; 0.2 ; 0.3]"
printf("x == [%f ; %f ; %f]\n", x[0], x[1], x[2]);

---

int
oski_MatTransMatMult(const **oski_matrix_t A_tunable**, **oski_ataop_t opA**,
oski_value_t alpha, const **oski_vecview_t x_view**,
oski_value_t beta, **oski_vecview_t y_view**, **oski_vecview_t t_view**);

Computes $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$, where $\text{op}(A) \in \{AA^T, A^TA, AA^H, A^HA\}$. Also optionally computes $t \leftarrow A \cdot x$ if $\text{op}(A) \in \{A^TA, A^HA\}$, $t \leftarrow A^T \cdot x$ if $\text{op}(A) = AA^T$, or $t \leftarrow A^H \cdot x$ if $\text{op}(A) = AA^H$, at the caller's request.

Parameters:
A_tunable [input] **A_tunable** is valid.
An object for a matrix A.

opA [input] See Table 7 on page 19.
Specifies $\text{op}(A)$.

alpha, **beta** [input]
The scalar constants α, β, respectively.

x_view [input] **x_view** is valid.
View object for a (multi)vector x.

y_view [input/output] **y_view** is valid.
View object for a (multi)vector y.

t_view [output] **t_view** may be valid or INVALID_MAT.
An optional view object for a (multi)vector t.

Actions and Returns:
Returns an error code and leaves y (and t, if specified) unchanged on error. Otherwise, returns 0 and computes $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$. On a 0-return, also computes t if **t_view** is specified and not equal to INVALID_MAT.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

Example:
// Let A_tunable be an object corresponding to the sparse lower triangular matrix
// shown in Figure 1 on page 8, and let $x^T = (.1 \quad .2 \quad .3)$. The following code computes
// $t \leftarrow A \cdot x$, and $y \leftarrow A^TA \cdot x$.

// Set-up vectors
double x[] = { .1, .2, .3 };
oski_vecview_t x_view = **oski_CreateVecView**(x, 3, STRIDE_UNIT);

double t[] = { −1, −1, −1 };
oski_vecview_t t_view = **oski_CreateVecView**(t, 3, STRIDE_UNIT);

double y[] = { 1, 1, 1 };

```
oski_vecview_t y_view = oski_CreateVecView( y, 3, STRIDE_UNIT );

// Execute kernel: $t \leftarrow A \cdot x, y \leftarrow A^T A \cdot x$
oski_MatTransMatMult( A_tunable, OP_AT_A, 1, x_view, 0, y_view, t_view );

// Print results. Should display
//     "t == [ 0.1 ; 0 ; 0.35 ];"
//     "y == [ 0.275 ; 0 ; 0.35 ];"
printf( "t == [ %f ; %f ; %f ];\n", t[0], t[1], t[2] );
printf( "y == [ %f ; %f ; %f ];\n", y[0], y[1], y[2] );
```

---

int
oski_MatMultAndMatTransMult(const **oski_matrix_t** **A_tunable**,
  oski_value_t **alpha**, const **oski_vecview_t** **x_view**,
  oski_value_t **beta**, **oski_vecview_t** **y_view**,
  oski_matop_t **opA**,
  oski_value_t **omega**, const **oski_vecview_t** **w_view**,
  oski_value_t **zeta**, **oski_vecview_t** **z_view**);

Computes $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$ and $z \leftarrow \omega \cdot \text{op}(A) \cdot x + \zeta \cdot z$, where $\text{op}(A) \in \{A, A^T, A^H\}$.

Parameters:

A_tunable [input] — **A_tunable** is valid.
An object for a matrix A.

alpha, **beta**, **omega**, **zeta** [input]
The scalar constants $\alpha, \beta, \omega, \zeta$, respectively.

opA [input] — See Table 6 on page 19.
Specifies $\text{op}(A)$.

x_view, **w_view** [input] — **x_view**, **w_view** are valid.
View objects for (multi)vectors x and w, respectively.

y_view, **z_view** [input/output] — **y_view**, **z_view** are valid.
View objects for (multi)vectors y and z, respectively.

Actions and Returns:
If A, x, and y have compatible dimensions, and if $\text{op}(A)$, w, and z have compatible dimensions, then this routine computes $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$ and $z \leftarrow \omega \cdot \text{op}(A) \cdot w + \zeta \cdot z$ and returns 0. Otherwise, returns an error code and takes no action.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

Example:

```
// Let A_tunable be a matrix object for the sparse lower triangular matrix $A$ shown in
// Listing 1 on page 8, and let $x^T = (.1 \quad .2 \quad .3)$. This example computes
//     $y \leftarrow A \cdot x$ and $z \leftarrow A^T \cdot x$.

double x[] = { .1, .2, .3 };
oski_vecview_t x_view = oski_CreateVecView( x, 3, STRIDE_UNIT );

double y[] = { −1, −1, −1 };
oski_vecview_t y_view = oski_CreateVecView( y, 3, STRIDE_UNIT );
```

```
double z[] = { 1, 1, 1 };
oski_vecview_t z_view = oski_CreateVecView( z, 3, STRIDE_UNIT );

// Compute y ← A· x and z ← A^T · x.
oski_MatMult_and_MatTransMult( A_tunable, 1, x_view, 0, y_view,
    OP_TRANS, 1, x_view, 0, z_view );

// Print results. Should print
//     "y == [ 0.1 ; 0 ; 0.35 ];"
//     "z == [ -0.15 ; 0.2 ; 0.3 ];"
printf( "y == [ %f ; %f ; %f ];", y[0], y[1], y[2] );
printf( "z == [ %f ; %f ; %f ];", z[0], z[1], z[2] );
```

---

int
oski_MatPowMult(const **oski_matrix_t A_tunable**, **oski_matop_t opA**, int **power**,
oski_value_t alpha, const **oski_vecview_t x_view**,
oski_value_t beta, **oski_vecview_t y_view**, **oski_vecview_t T_view**);

Computes a power of a matrix times a vector, or $y \leftarrow \alpha \cdot \mathrm{op}(A)^{\rho} \cdot x + \beta \cdot y$. Also optionally computes $T = \begin{pmatrix} t_1 & \cdots & t_{\rho-1} \end{pmatrix}$, where $t_k \leftarrow \mathrm{op}(A)^{k} \cdot x$ for all $1 \leq k < \rho$.

A_tunable [input] — **A_tunable** is valid.
An object for a matrix A. If $\rho > 1$, then A must be square.

opA [input] — See Table 6 on page 19.
Specifies $\mathrm{op}(A)$.

power [input] — **power** ≥ 0
Power ρ of the matrix A to apply.

alpha, **beta** [input]
The scalar constants α, β, respectively.

x_view [input] — **x_view** is a valid, single vector.
View object for the vector x.

y_view [input/output] — **y_view** is valid, single vector.
View object for the vector y.

T_view [output] — **T_view** is a valid multivector view of at least $\rho - 1$ vectors, orNULL.
If non-NULL, **T_view** is a view object for the multivector $T = \begin{pmatrix} t_1 & \cdots & t_{\rho-1} \end{pmatrix}$.

Actions and Returns:
Let A be an $n \times n$ matrix. The vectors x and y must be single vectors of length n. If T is specified via a valid **T_view** object, then T must have dimensions $n \times (\rho - 1)$. If all these conditions are satisfied, then this routine computes $y \leftarrow A^{\rho} \cdot x + \beta \cdot y$, $t_k \leftarrow A^{k} \cdot x$ for all $1 \leq k < \rho$ (if appropriate), and returns 0. Otherwise, no action is taken and an error code is returned.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

Example:

// First create a matrix $A = \begin{pmatrix} .25 & 0 & 0 \\ 0 & .75 & 0 \\ .75 & .25 & 1 \end{pmatrix}$ in CSR format using 1-based indices.

```
int  Aptr[] = { 1, 2, 3, 6 };        // 1-based
int  Aind[] = { 1, 2, 1, 2, 3 };   // 1-based
double Aval[] = { .25, .75, .75, .25, 1 };
oski_matrix_t A_tunable = oski_CreateMatCSR( Aptr, Aind, Aval, 3, 3, SHARE_INPUTMAT, 0 );

// Create a vector x^T = (1   1   1).
double x[] = { 1, 1, 1 };
oski_vecview_t x_view = oski_CreateVecView( x, 3, STRIDE_UNIT );

// Result vector y
double y[] = { −1, −1, −1 };
oski_vecview_t y_view = oski_CreateVecView( y, 3, STRIDE_UNIT );

// Storage space to keep intermediate vectors, T = (t_1   t_2).
// Initially, let t_1^T = (.1   .1   .1), and t_2^T = (.2   .2   .2).
double T[] = { .1, .1, .1, .2, .2, .2 };   // in column-major order
oski_vecview_t T_view = oski_CreateMultiVecView( T, 3, 2, LAYOUT_COLMAJ, 3 );

// Compute y ← A^3 · x and the intermediate vectors t_1, t_2.
oski_MatPowMult( A_tunable, OP_NORMAL, 3, 1.0, x_view, 0.0, y_view, T_view );

// Print results:
//      "t1 = A*x = [ 0.25 ; 0.75 ; 2 ];"
//      "t2 = Aˆ2*x = [ 0.0625 ; 0.5625 ; 2.375 ];"
//      "y = Aˆ3*x = [ 0.015625 ; 0.421875 ; 2.5625 ];"
printf( "t1 = A*x = [ %f ; %f ; %f ];\n", T[0], T[1], T[2] );
printf( "t2 = Aˆ2*x = [ %f ; %f ; %f ];\n", T[3], T[4], T[5] );
printf( "y = Aˆ3*x = [ %f ; %f ; %f ];\n", y[0], y[1], y[2] );
```

B.4 Tuning

```
int
oski_SetHintMatMult( oski_matrix_t A_tunable, oski_matop_t opA,
    oski_value_t alpha, const oski_vecview_t x_view,
    oski_value_t beta, const oski_vecview_t y_view,
    int num_calls );
```

Workload hint for the kernel operation **oski_MatMult** which computes $y \leftarrow \alpha \cdot \mathrm{op}(A) \cdot x + \beta \cdot y$, where $\mathrm{op}(A) \in \{A, A^T, A^H\}$.

Parameters:

A_tunable [input/output] — **A_tunable** is valid.
An object for a matrix A.

opA [input] — See Table 6 on page 19.
Specifies op(A).

alpha, **beta** [input]
Scalar constants α, β, respectively.

x_view, **y_view**[input] — Vectors are valid or symbolic (see Table 11 on page 22).
View object for a (multi)vector x and y, respectively..

num_calls [input] — **num_calls** is non-negative or symbolic (see Table 10 on page 22).
The number of times this kernel will be called with these arguments.

Actions and Returns:
Registers the workload hint with **A_tunable** and returns 0 only if the dimensions of op(A), x, and y are compatible. Otherwise, returns an error code.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

Example:
See Listing 2 on page 11.

---

int
oski_SetHintMatTrisolve(**oski_matrix_t** **T_tunable**, **oski_matop_t** **opT**,
    oski_value_t **alpha**, const **oski_vecview_t** **x_view**,
    int **num_calls**);

Workload hint for the kernel operation **oski_MatTrisolve** which computes $x \leftarrow \alpha \cdot \text{op}(T)^{-1} \cdot x$, where T is a triangular matrix.

Parameters:
T_tunable [input/output] **T_tunable** is valid, square, and triangular.
Matrix object for an $n \times n$ upper or lower triangular matrix T.

opT [input] See Table 6 on page 19.
Specifies op(T).

alpha [input]
Scalar constant α.

x_view [input] **x_view** is valid or symbolic (see Table 11 on page 22).
View object for a (multi)vector x.

num_calls [input] **num_calls** is non-negative or symbolic (see Table 10 on page 22).
The number of times this kernel will be called with these arguments.

Actions and Returns:
Registers the workload hint with **A_tunable** and returns 0 only if the dimensions of op(T) and x have compatible dimensions. Otherwise, returns an error code.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

---

int
oski_SetHintMatTransMatMult(**oski_matrix_t** **A_tunable**, **oski_ataop_t** **opA**,
    oski_value_t **alpha**, const **oski_vecview_t** **x_view**,
    oski_value_t **beta**, const **oski_vecview_t** **y_view**,
    [const **oski_vecview_t** **t_view**,]
    int **num_calls**);

Workload hint for the kernel operation **oski_MatTransMatMult** which computes $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$, where $\text{op}(A) \in \{AA^T, A^TA, AA^H, A^HA\}$, and also optionally computes $t \leftarrow A \cdot x$ if $\text{op}(A) \in \{A^TA, A^HA\}$, $t \leftarrow A^T \cdot x$ if $\text{op}(A) = AA^T$, or $t \leftarrow A^H \cdot x$ if $\text{op}(A) = AA^H$.

Parameters:

A_tunable [input/output] **A_tunable** is valid.
An object for a matrix A.

opA [input] See Table 7 on page 19.
Specifies op(A).

alpha, **beta** [input]
The scalar constants α, β, respectively.

x_view, **y_view** [input] **x_view**, **y_view** are valid or symbolic (see Table 11 on page 22).
View objects for (multi)vector objects x, y, respectively. for a (multi)vector x.

t_view [input] May be valid, symbolic, or INVALID_MAT.
An optional view object for a (multi)vector t.

num_calls [input] **num_calls** is non-negative or symbolic (see Table 10 on page 22).
The number of times this kernel will be called with these arguments.

Actions and Returns:
Registers the workload hint with **A_tunable** and returns 0 only if the argument dimensions are compatible. Otherwise, returns an error code.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

---

int
oski_SetHintMatMultAndMatTransMult(**oski_matrix_t** **A_tunable**,
  oski_value_t **alpha**, const **oski_vecview_t** **x_view**,
  oski_value_t **beta**, const **oski_vecview_t** **y_view**,
  oski_matop_t **opA**,
  oski_value_t **omega**, const **oski_vecview_t** **w_view**,
  oski_value_t **zeta**, const **oski_vecview_t** **z_view**,
  int **num_calls**);

Workload hint for the kernel operation **oski_MatMultAndMatTransMult** which computes $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$ and $z \leftarrow \omega \cdot \text{op}(A) \cdot x + \zeta \cdot z$, where $\text{op}(A) \in \{A, A^T, A^H\}$.

Parameters:
A_tunable [input/output] **A_tunable** is valid.
An object for a matrix A.

alpha, **beta**, **omega**, **zeta** [input]
The scalar constants $\alpha, \beta, \omega, \zeta$, respectively.

opA [input] See Table 6 on page 19.
Specifies op(A).

x_view, **y_view**, **w_view**, **z_view** [input] Vectors are valid or symbolic (see Table 11 on page 22).
View objects for (multi)vectors x, y, w, and z, respectively.

num_calls [input] **num_calls** is non-negative or symbolic (see Table 10 on page 22).
The number of times this kernel will be called with these arguments.

Actions and Returns:

If A, x, and y have compatible dimensions, and if $\text{op}(A)$, w, and z have compatible dimensions, then this routine registers the workload hint with **A_tunable** and returns 0. Otherwise, returns an error code.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

---

int
oski_SetHintMatPowMult(**oski_matrix_t** **A_tunable**, **oski_matop_t** **opA**, int **power**,
    oski_value_t **alpha**, const **oski_vecview_t** **x_view**,
    oski_value_t **beta**, const **oski_vecview_t** **y_view**,
    const **oski_vecview_t** **T_view**,
    int **num_calls**);

Workload hint for the kernel operation **oski_MatPowMult** which computes a power of a matrix times a vector, or $y \leftarrow \alpha \cdot \text{op}(A)^{\rho} \cdot x + \beta \cdot y$. Also optionally computes $T = (t_1 \quad \cdots \quad t_{\rho-1})$, where $t_k \leftarrow \text{op}(A)^{k} \cdot x$ for all $1 \leq k < \rho$.

A_tunable [input/output] — **A_tunable** is valid and square.
An object for a matrix A.

opA [input] — See Table 6 on page 19.
Specifies $\text{op}(A)$.

power [input] — **power** $\geq$0
Power ρ of the matrix A to apply.

alpha, **beta** [input]
The scalar constants α, β, respectively.

x_view, **y_view** [input] — Vectors are valid or symbolic (see Table 11 on page 22) single vectors.
View objects for the vectors x, y.

T_view [input] — A valid or symbolic multivector, or INVALID_MAT.
If not equal to INVALID_MAT, **T_view** is either a view object for the multivector $T = (t_1 \quad \cdots \quad t_{\rho-1})$, or SYMBOLIC_MULTIVEC.

num_calls [input] — **num_calls** is non-negative or symbolic (see Table 10 on page 22).
The number of times this kernel will be called with these arguments.

Actions and Returns:
Registers the workload hint with **A_tunable** and returns 0 if the operand dimensions are compatible. Otherwise, returns an error code.

Error conditions and actions:
Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MAT, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

---

int
oski_SetHint(**oski_matrix_t** **A_tunable**, oski_tunehint_t **hint** [, ...]);

Register a hint about the matrix structure with a matrix object.

Parameters:
A_tunable [input/output] — **A_tunable** is valid.

Matrix object for which to register a structural hint.

hint [input] See Table 9 on page 21.
User-specified structural hint. This hint may be followed by optional arguments, as listed and typed in Table 9 on page 21.

Actions and Returns:
Returns 0 if the hint is recognized and **A_tunable** is valid, or an error code otherwise.

Error conditions and actions:
Possible error conditions include an invalid matrix object [ERR_BAD_MAT], or specifying a hint with the wrong number of hint arguments [ERR_BAD_HINT_ARG].

Example:
See Listing 2 on page 11.

---

int
oski_TuneMat(**oski_matrix_t** **A_tunable**);

Tune the matrix object using all hints and implicit profiling data.

Parameters:
A_tunable [input/output] **A_tunable** is valid.
Matrix object to tune.

Actions and Returns:
Returns a non-negative status code whose possible values are defined by the constants listed in Table 12 on page 23, or an error code otherwise.

Error conditions and actions:
Possible error conditions include providing an invalid matrix [ERR_BAD_MAT].

Example:
See Listing 2 on page 11 and Listing 3 on page 12.

B.5 Permutations

---

int
oski_IsMatPermuted(const **oski_matrix_t** **A_tunable**);

Checks whether a matrix has been tuned by reordering.

Parameters:
A_tunable [input] **A_tunable** is valid.
A matrix object corresponding to some matrix A.

Actions and Returns:
Returns 1 if **A_tunable** has been tuned by reordering. That is, if tuning produced a representation $A = P_r^T \cdot \hat{A} P_c$, where either P_r or P_c is not equal to the identity matrix I, then this routine returns 1. If $P_r = P_c = I$, then this routine returns 0. Returns an error code on error.

Error conditions and actions:
Possible error conditions include providing an invalid matrix [ERR_BAD_MAT].

Example:
See Listing 4 on page 25.

---

const **oski_matrix_t**
oski_ViewPermutedMat(const **oski_matrix_t** **A_tunable**);

Given a matrix A, possibly reordered during tuning to the form $\hat{A} = P_r \cdot A \cdot P_c^T$, returns a read-only object corresponding to $\hat{A}$.

Parameters:
A_tunable [input] **A_tunable** is valid.
A matrix object corresponding to some matrix A.

Actions and Returns:
Returns a read-only matrix object representing $\hat{A}$. This return is exactly equal to **A_tunable** if the matrix is not reordered, *i.e.*, if $P_r = P_c = I$, the identity matrix. Returns INVALID_MAT on error.

Error conditions and actions:
Possible error conditions include providing an invalid matrix [ERR_BAD_MAT].

Example:
See Listing 4 on page 25.

---

const **oski_perm_t**
oski_ViewPermutedMatRowPerm(const **oski_matrix_t** **A_tunable**);

Given a matrix A, possibly reordered during tuning to the form $\hat{A} = P_r \cdot A \cdot P_c^T$, returns a read-only object corresponding to P_r.

Parameters:
A_tunable [input] **A_tunable** is valid.
A matrix object corresponding to some matrix A.

Actions and Returns:
Returns a read-only permutation object representing P_r. This return is exactly equal to PERM_IDENTITY if the matrix is not reordered, *i.e.*, if $P_r = P_c = I$, the identity matrix. Returns INVALID_PERM on error.

Error conditions and actions:
This routine calls the error handler and returns INVALID_MAT if any argument preconditions are not satisfied.

Example:
See Listing 4 on page 25.

---

const **oski_perm_t**
oski_ViewPermutedMatColPerm(const **oski_matrix_t** **A_tunable**);

Given a matrix A, possibly reordered during tuning to the form $\hat{A} = P_r \cdot A \cdot P_c^T$, returns a read-only object corresponding to P_c.

Parameters:
A_tunable [input] **A_tunable** is valid.
A matrix object corresponding to some matrix A.

Actions and Returns:
Returns a read-only permutation object representing P_c. This return is exactly equal to PERM_IDENTITY if the matrix is not reordered, *i.e.*, if $P_r = P_c = I$, the identity matrix. Returns INVALID_PERM on error.

Error conditions and actions:
This routine calls the error handler and returns INVALID_MAT if any argument preconditions are not satisfied.

Example:
See Listing 4 on page 25.

---

int
oski_PermuteVecView(const **oski_perm_t** P, **oski_matop_t opP**, **oski_vecview_t x_view**);

Permute a vector view object, *i.e.*, computes $x \leftarrow \text{op}(P) \cdot x$.

Parameters:
P [input] P is valid.
An object corresponding to some permutation, P.

opP [input]
Specifies op(P).

x_view [input/output] **x_view** is valid.
The view object corresponding to the (multi)vector x.

Actions and Returns:
Permutes the elements of x and returns 0. On error, returns an error code and leaves **x_view** unchanged.

Error conditions and actions:
Possible error conditions include providing an invalid permutation [ERR_BAD_PERM] or vector [ERR_BAD_VECVIEW].

Example:
See Listing 4 on page 25.

B.6 Saving and restoring tuning transformations

---

char ∗
oski_GetMatTransforms(const **oski_matrix_t A_tunable**);

Returns a string representation of the data structure transformations that were applied to the given matrix during tuning.

Parameters:
A_tunable [input] **A_tunable** is valid.
The matrix object to which from which to extract the specified data structure transformations.

Actions and Returns:
Returns a newly-allocated string representation of the transformations that were applied to the given matrix during tuning, or NULL on error. The user must deallocate the returned string by an appropriate call to the C **free()** routine. Returns NULL on error.

Error conditions and actions:

Possible error codes include providing an invalid matrix [ERR_BAD_MAT].

Example:
See Listing 5 on page 26.

---

int
oski_ApplyMatTransforms(const **oski_matrix_t** **A_tunable**, const char∗ **xforms**);

Replace the current data structure for a given matrix object with a new data structure specified by a given string.

Parameters:
A_tunable [input] **A_tunable** is valid.
The matrix object to which to apply the specified data structure transformations.

xforms [input]
A string representation of the data structure transformations to be applied to the matrix represented by **A_tunable**. The conditions **xforms** == NULL and **xforms** equivalent to the empty string are both equivalent to requesting no changes to the data structure.

Actions and Returns:
Returns 0 if the transformations were successfully applied, or an error code otherwise. On success, the data structure specified by **xforms** *replaces* the existing tuned data structure if any.

Error conditions and actions:
Possible error conditions include an invalid matrix [ERR_BAD_MAT], a syntax error while processing **xforms** [ERR_BAD_SYNTAX], and an out-of-memory condition [ERR_OUT_OF_MEMORY].

Example:
See Listing 6 on page 26.

B.7 Error handling

---

oski_errhandler_t
oski_GetErrorHandler(void);

Returns a pointer to the current error handling routine.

Actions and Returns:
Returns a pointer to the current error handler, or NULL if there is no registered error handler.

---

oski_errhandler_t
oski_SetErrorHandler(**oski_errhandler_t** **new_handler**);

Changes the current error handler.

Parameters:
new_handler [input] A valid error handling routine, or NULL.
Pointer to a new function to handle errors.

Actions and Returns:
This routine changes the curent error handler to be **new_handler** and returns a pointer to the previous error handler.

void
oski_HandleErrorDefault(int **error_code**,
    const char* **message**, const char* **source_filename**, unsigned long **line_number**,
    const char* **format_string**, . . .);

The default error handler, called when one of the OSKI routines detects an error condition.

Parameters:
message [input]
A descriptive string message describing the error or its context. The string **message** == NULL if no message is available.

source_filename [input]
The name of the source file in which the error occurred, or NULL if not applicable.

line_number [input]
The line number at which the error occurred, or a non-positive value if not applicable.

format_string [input] A printf-compatible format string.
A formatting string for use with the printf routine. This argument (and any remaining arguments) may be passed to a printf-like function to provide any supplemental information.

Actions and Returns:
This routine dumps a message describing the error to standard error.

C BeBOP Library Integration Notes

The following subsections discuss integration of OSKI with specific higher-level libraries and applications.

C.1 PETSc

The PETSc (Portable Extensible Toolkit for Scientific computing) library provides a portable interface for distributed iterative sparse linear solvers based on MPI, and is a primary integration target of our interface [6, 5]. PETSc is written in C in an object-oriented style which defines an abstract matrix interface (type Mat); specific matrix formats are concrete types derived from this interface that implement the interface. Available formats at the time of this writing include serial CSR (type MatAIJ), distributed CSR (MatMPIAIJ), serial and distributed block compressed sparse row format (MatBAIJ and MatMPIBAIJ, respectively, for square block sizes up to 8×8), and a matrix-free "format" (MatShell, implemented using callbacks to user-defined routines) among others.

The simplest way to integrate OSKI into PETSc is to define a new concrete type (say, MatOSKI) based on the distributed compressed sparse row format MatMPIAIJ, with the following modifications:

- PETSc distributes blocks of consecutive rows of the matrix across the processors. Internally, each processor stores its rows in two packed 3-array 0-based CSR data structures (Appendix A on page 38): one for a diagonal block, and one for all remaining elements. PETSc determines the distribution and sets up the corresponding data structures when the user calls a special matrix assembly routine. The MatOSKI type would store two additional handles on each processor corresponding to the representation.

- Since the abstract matrix interface defines a large number of possible "methods," each BeBOP handle would be created using the shared copy mode (SHARE_INPUTMAT, as discussed in Section 3.2.1 on page 14) so that not all methods would have to be specialized initially. The cost of this arrangement is that there may be two copies of the matrix (the local CSR data structure and a tuned data structure).

- At a minimum, an initial implementation of MatOSKI should implement the following methods defined by Mat: matrix assembly, get/set non-zero values, and SpMV kernel calls.

- Tuning could be performed in the implicit self-profiling style described in Section 2.3 on page 12, since PETSc will not necessarily *a priori* which solver the user will call (and therefore it will not known the number of SpMV operations). The call to **oski_TuneMat** could be inserted after each SpMV call since such calls will be low-overhead calls until tuning occurs.

- Symmetric matrices in PETSc are stored in full storage format, though a special tag identifies the matrix as symmetric. For MatOSKI, we may specify that the diagonal block is symmetric by providing the appropriate hint at handle creation time (see Table 3 on page 17).

- The default error handler for each OSKI handle should be replaced with a routine that prints an error message to PETSc's error log.

- We recommend that additional routines and command-line options be created, conforming to PETSc's calling style, to provide access to the following OSKI interface features: always tuning at matrix assembly, and saving and restoring tuning transformations. Such routines and options would act as "no-ops" if the user called them on a matrix not of type MatOSKI.

Users can use the usual PETSc mechanisms to select this type either on the command-line at run-time, or by explicitly requesting it via an appropriate function call in their applications. For most users, this amounts to a one-line change in their source code.

C.2 MATLAB*P

MATLAB*P is a research prototype system providing distributed parallel extensions to MATLAB [43]. Internally, matrices are stored in CSC format.

Since MATLAB*P is intended for use interactively, the amount of time available for tuning may be limited. Therefore, we recommend using OSKI in the implicit self-profiling mode described in Section 2.3 on page 12.