The Optimized Sparse Kernel Interface (OSKI) Library* User's Guide for Version 1.0.1h

Richard Vuduc James W. Demmel
Katherine A. Yelick
Berkeley Benchmarking and OPtimization (BeBOP) Group
University of California, Berkeley
http://bebop.cs.berkeley.edu/oski

June 25, 2007

OSKI is based on research supported in part by the National Science Foundation under NSF Cooperative Agreement No. ACI-9813362, NSF Cooperative Agreement No. ACI-9619020, the Department of Energy under DOE Grant No. DE-FC02-01ER25478, and gifts from HP and Intel. This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by DOE under Contract No. DE-AC05-00OR22725, the High Performance Computing Research Facility, Mathematics and Computer Science Division, Argonne National Laboratory, the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory, the University of Electro-Communications in Tokyo, Japan, the Dept. of Biomedical Informatics at Ohio State University, and the OpenPower Project at the University of Augsburg, Germany. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Contents

List of Tables

Listings

1 What is OSKI and Who Should Use It?

The Optimized Sparse Kernel Interface (OSKI) [1] is a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices, for use by the developers of solver libraries and scientific and engineering applications. These kernels include sparse matrix-vector multiply and sparse triangular solve, among others. "Tuning" means choosing a data structure and corresponding kernel implementation that both (a) compactly represents the particular sparse matrix, and (b) has an efficient implementation on the target machine. The primary aim of OSKI is to hide the complex decision-making process needed to tune, while also exposing the steps and potentially non-trivial costs of tuning at run-time. The interface also allows for optional continuous profiling and periodic re-tuning, as well as user inspection and control of the tuning process. OSKI provides functionality essentially at the level of the Basic Linear Algebra Subroutines (BLAS) [3, 6], and we assume a user who has a basic familiarity with the BLAS. The current version of OSKI targets uniprocessors machines based on cache-based superscalar microprocessors, although we are actively extending OSKI for vector architectures, SMPs, and distributed memory machines.

OSKI is part of on-going work by the Berkeley Benchmarking and OPtimization (BeBOP) group, an active research program on automatic performance tuning and analysis at the University of California, Berkeley. OSKI is based in large part on the SPARSITY framework for automatically tuning sparse matrix kernels [14]. For information on the philosophy of the OSKI design and pointers to the related research, see the OSKI Design Document [20], a draft of which is available at:

http://bebop.cs.berkeley.edu/oski

How to read this document

The fastest way to get started is to look at the material in Sections 2–4, which contains enough information to build and install OSKI and to write your first test program. If you run into problems, see Section 6 on page 30 for debugging hints, or post a question in the online forum through the OSKI home page.

2 Installation

This section describes how to compile and install OSKI 1.0.1h from the source. There may also be pre-compiled binaries for your platform; check the OSKI home page.

Automatic tuning in OSKI happens in two phases: an installation time phase which benchmarks your machine, and a run-time phase when you call OSKI with a particular matrix. You will need to run a benchmark during installation, but this step is automated and should not require any manual intevention on your part.

2.1 What you will need to get started

To compile and install OSKI 1.0.1h, you will need the following:

- An ANSI C compiler.
- An environment providing standard UNIX tools, such as `make`, `grep`, `awk`, and `sed`. OSKI has been tested in the following environments: Linux, FreeBSD, NetBSD, AIX,

[1] OSKI is also the name of the U.C. Berkeley mascot. Go Bears!

IRIX, Darwin, and Cygwin. Microsoft Windows platforms are supported only through Cygwin.

You'll also need to think about the following issues:

- **What integer and floating-point precisions do you need?**

  Sparse matrices are stored using both scalar integer indices and floating-point values, and you can build different versions of OSKI with various combinations of types. OSKI currently supports supports the C language `int` and `long` (usually 32-bit and 64-bit values, respectively) for the integer types, and for floating-point values supports single-precision real using `float`, double-precision real using `double`, and complex-valued versions of single- and double-precision. By default, the installation process builds `int-double`. You specify the combinations of types you need during the **Configuration** step below.

- **In what directory will you install OSKI?**

  OSKI installs itself in the subdirectories of `/usr/local` by default, but you can override this. In all of our examples, we will use `${OSKIDIR}` as a placeholder for the directory of your choice. The OSKI files are installed in:

  ```
  ${OSKIDIR}/bin
  ${OSKIDIR}/lib/oski
  ${OSKIDIR}/include/oski
  ```

- **Do you want to build static or shared libraries?**

  The default installation builds both static and shared libraries, except on Cygwin where only static libraries are built.[2]

  We recommend that, when possible, you build and use shared libraries for two reasons. First, the entire OSKI library can be quite large and shared libraries will help reduce the size of your executable. Second, we designed OSKI to allow new extensions to be "plugged in" as dynamically loaded modules.

- **Do you want OSKI to use any support libraries?**

  OSKI can take advantage of additional libraries that may be available for your system, such as a highly optimized dense BLAS library, or PAPI for timing/benchmarking. For a list, see Section 2.3.3 on page 7.

2.2 How to install OSKI

Follow these steps to compile and install OSKI. We show sample command-lines for sh/csh-compatible shells, where `%` denotes the command-line prompt.

1. **Download** `oski-1.0.1h.tar.gz` (or `oski-1.0.1h.tar.bz2`) from the OSKI home page.

2. **Unpack** the distribution.

   ```
   %  gunzip -c oski-1.0.1h.tar.gz | tar xvf -
   ```

   This command unpacks the OSKI distribution into a subdirectory named `oski-1.0.1h`.

[2] It is possible to create Windows-style DLLs using the `--enable-shared` configure flag, but we currently consider their use to be experimental. Proceed with caution.

3. **Configure** OSKI for your platform:

```
%  mkdir build-1.0.1h
%  cd build-1.0.1h
%  ../oski-1.0.1h/configure --prefix=${OSKIDIR}
```

   As done in this example, we strongly recommend building OSKI in a directory separate from the source tree you unpacked in Step 2. Here we use a directory named `build-1.0.1h`.

   The configure script tries to detect your operating system and CPU, and chooses a compiler and default compiler optimization flags accordingly. These choices will be displayed when the script finishes. If you wish to specify these flags yourself, or if `configure` is unable to choose a compiler or default flags, see Section 2.3.1 on the following page.

   This step as shown builds a `int-double` precision library by default. To build other type combinations, see Section 2.3.2 on the next page.

4. **Compile** OSKI:

```
%  make
```

   Compilation times vary widely across platforms, and could take anywhere between half-an-hour to a several hours. This is a good time for that much needed coffee break.

5. **Benchmark** OSKI.

```
%  make benchmarks
```

   Like compilation, benchmark times vary widely across machines. Allow for approximately half-an-hour to an hour.

   The sufficiently curious may view the raw benchmark data located in `bench/*.dat` files in the build tree. Section 6.3 on page 30 describes these files in additional detail.

6. (OPTIONAL) **Test** OSKI.

```
%  make check
```

   This step runs your build of OSKI through an extensive series of tests. The test battery includes all precision combinations selected at `configure`-time. These tests are designed primarily for developer regression testing, and could take an hour or more to run.

7. **Install** OSKI.

```
%  make install
```

   This step installs all of the OSKI files and benchmarking data into the subdirectories of `${OSKIDIR}` specified in Section 2.1 on page 4.

That completes installation of OSKI. You should be ready for the first example in Section 3 on page 8.

2.3 Customizing your OSKI build using `configure`

The **Configure** step can be customized in many ways. The most commonly used options are discussed below. For a complete list, run (from the build tree in our previous example):

```
%  ../oski-1.0.1h/configure --help
```

2.3.1 Overriding the default compiler and/or flags

To specify the compiler and/or compiler flags yourself, define the `CC` and/or `CFLAGS` environment variables accordingly. For example, to use the Intel C compiler to build OSKI specifically for a Pentium-M machine:

```
%  env CC=icc CFLAGS='-O3 -xB' ../oski-1.0.1h/configure ...
```

We very much welcome your contributions and suggestions if you find compiler and optimization flag combinations that improve on the defaults.

The **Configure** step will try to auto-detect a Fortran 77 compiler, so that OSKI can later check for and link against a native BLAS library. If you do not want or have a Fortran compiler and run into configuration problems, try specifying the `--disable-fortran` flag. To override the F77 compiler or F77 compiler flags, use the `F77` or `FFLAGS` environment variables, as with `CC`/`CFLAGS` above.

2.3.2 Selecting other or additional scalar type precisions

By default, OSKI uses your C compiler's `int` data type to store integer indices, and `double` for floating-point values. Select different combinations by adding the `configure`-time flag, `--enable-<INT>-<VAL>`, where valid `<INT>` types are `int` and `long`, and valid floating-point value types are `single`, `double`, `scomplex`, and `dcomplex`. You may specify any number of combinations, as well as disable the default.

For example, to disable the default `int-double` version, and instead build both the `int`-single-complex and `long`-double-complex versions, run

```
%  ../oski-1.0.1h/configure --prefix=${OSKIDIR} --disable-int-double \
              --enable-int-scomplex --enable-long-dcomplex
```

To use these types in your application, see Section 5.1 on page 16.

2.3.3 Optional tools

OSKI can use any of the freely-available, highly-tuned dense BLAS libraries, such as ATLAS [23] or Goto's BLAS [9], or commercial hardware vendor implementations. The `configure` script will attempt to detect their presence, but you can also manually specify how to link against them with the `--with-blas=<LINK-FLAGS>` option. For example,

```
%  ../oski-1.0.1h/configure --prefix=${OSKIDIR} \
       --with-blas='-L/my-blas-path -lmy_blas'
```

asks OSKI to link against the `libmy_blas.a` library in the `/my-blas-path` directory.

To obtain high-resolution timing, OSKI internally uses the hardware cycle counter reader subpackage of FFTW 3.0.1 [8]. Alternatively, you can ask OSKI to use your local installation of the Performance API (PAPI) cycle counter reader using the `configure`-time option, `--with-papi`, to specify the appropriate link flags. For example, to link against the copy of PAPI in the `papi-3.0.8/lib` subdirectory of your home directory:

```
%  ../oski-1.0.1h/configure --with-prefix=${OSKIDIR} \
        --with-papi="-L${HOME}/papi-3.0.8/lib -lpapi"
```

We recommend that you use PAPI only if you encounter an error with the timing package during installation.

3 Using OSKI: A First Example

This section introduces the C version[3] of OSKI by example. 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. In addition to showing OSKI's basic usage and providing you with a first example to test your OSKI installation, this example illustrates how you can gradually migrate your existing application to use OSKI, provided that application uses "standard" array representations of sparse matrices and dense vectors.

Listing 1 on the following page shows a simple example C program that computes one sparse matrix-vector multiply (SpMV) on a 3×3 lower-triangular matrix A, and then prints the result. (This example is located in the `example/01` subdirectory of the OSKI source tree.) We explain Listing 1 on the next page below; to just build the program to test your installation, skip ahead to Section 3.5 on page 11. Calls to OSKI routines are shown in **bold-face**, OSKI "objects" are shown in **blue bold-face**, OSKI constants are shown red. Furthermore, this example assumes you have built the default `int-double` version of the library.

3.1 Initializing OSKI

To initialize the library, your application should include `oski/oski.h` (line 7) and call **oski_Init** (line 27). For symmetry, you can call **oski_Close** (line 44) at the end of your program to shut-down the library, although this step is optional.

3.2 Creating a matrix

The sparse matrix in Listing 1 on the next page is a 3×3 lower triangular matrix with all ones on the diagonal. The input matrix, here declared statically for simplicity in lines 15–17, is stored in a compressed sparse row (CSR) format 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 [17, 16, 1]. This particular example assumes the convention of 0-based indices and does not store the diagonal explicitly.

Lines 19–20 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 28–31) with the following arguments:

1. Arguments 1–3 specify the CSR arrays (line 28).
2. Arguments 4–5 specify the matrix dimensions (line 28).

[3] Fortran interfaces will be available soon.

Listing 1: **Using OSKI: A first example**. This example illustrates basic object creation and kernel execution in OSKI. 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 ← α · A· x + β · y, where
 * A = [1 0 0; −2 1 0; .5 0 1], x = [.25; .45; .65], and y is initially [1; 1; 1]
 * A is a sparse lower triangular matrix with a unit diagonal, and x, y are dense vectors.
 */

#include <stdio.h>
#include <oski/oski.h> /* Get OSKI bindings */

#define DIM 3 /* matrix dimension */
#define NUM_STORED_NZ 2 /* number of stored non-zero values */

int main()
{
    /* Sparse matrix A, in compressed sparse row (CSR) format */
    int Aptr[DIM+1] = { 0, 0, 1, 2 };
    int Aind[NUM_STORED_NZ] = { 0, 0 };
    double Aval[NUM_STORED_NZ] = { −2, 0.5 };
    /* Dense vectors, x, y, and scalar multipliers, α, β */
    double x[DIM] = { .25, .45, .65 };
    double y[DIM] = { 1, 1, 1 };
    double alpha = −1, beta = 1;
    /* OSKI matrix/vector objects */
    oski_matrix_t A_tunable;
    oski_vecview_t x_view, y_view;

    /* Initialize OSKI and create matrix */
    oski_Init ();
    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. */
    x_view = oski_CreateVecView( x, 3, STRIDE_UNIT );
    y_view = oski_CreateVecView( y, 3, STRIDE_UNIT );

    /* Perform matrix vector multiply, y ← α · A· x + β · y. */
    oski_MatMult( A_tunable, OP_NORMAL, alpha, x_view, beta, y_view );

    /* Clean−up interface objects and shut down OSKI */
    oski_DestroyMat( A_tunable );
    oski_DestroyVecView( x_view );
    oski_DestroyVecView( y_view );
    oski_Close();

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

3. The 6th argument to **oski_CreateMatCSR** (line 29) specifies one of two possible *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 41), 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 you may choose to free the arrays at any time. We discuss the semantics of both modes in detail in Section 5 on page 14. In this example, think of A_tunable as a wrapper around these arrays.

4. Arguments 7–10 tell the library how to interpret the CSR arrays (line 31). Argument 7 is a count that says the next 3 arguments are semantic properties needed to interpret the input matrix correctly. 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 will, 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 example uses the SHARE_INPUTMAT copy mode and performs no tuning, OSKI 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 you do not provide the optional arguments, the library assumes the defaults discussed in Section 5.2 on page 16.

3.3 Specifying the dense vectors

Dense vector objects of type oski_vecview_t, are always wrappers around user array representations (lines 34–35). 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 5.2.3 on page 19. These vector views help unify and simplify some of the kernel argument lists.

3.4 Calling sparse matrix-vector multiply

The argument lists to kernels, such as **oski_MatMult** for SpMV in this example (line 37), follow the conventions of the 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 41–43). However, since the user and library share the arrays underlying A_tunable, x_view, and y_view, you are responsible for deallocating these arrays (here, Aptr, Aind, Aval, x, and y).

Listing 2: **`Makefile` for the first example**. This `Makefile` builds shared and static library versions of Listing 1 on page 9.

```
1  # Builds the program in Listing 1 on page 9 on Linux systems.

   # Location of your copy of OSKI
   OSKI_DIR = /usr/local
   OSKIINCS_DIR = $(OSKI_DIR)/include
6  OSKILIBS_DIR = $(OSKI_DIR)/lib/oski

   # OSKI link flags
   OSKILIBS_SHARED = -Wl,-rpath -Wl,$(OSKILIBS_DIR) -L$(OSKILIBS_DIR) \
                     `cat $(OSKILIBS_DIR)/site-modules-shared.txt`
11 OSKILIBS_STATIC = -Wl,--whole-archive \
                     `cat $(OSKILIBS_DIR)/site-modules-static.txt` \
                     -Wl,--no-whole-archive

   CC = icc
16 CFLAGS = -I$(OSKI_DIR)/include -O3 -xB -g
   CLDFLAGS_SHARED = $(OSKILIBS_SHARED) -lm
   CLDFLAGS_STATIC = $(OSKILIBS_STATIC) -ldl -lm

   #-----------------------------------------
21 all : example1-shared example1-static

   example1-shared: example1.o
       $(CC) $(CFLAGS) -o $@ example1.o $(CLDFLAGS_SHARED)

26 example1-static: example1.o
       $(CC) $(CFLAGS) -o $@ example1.o $(CLDFLAGS_STATIC)

   .c.o:
       $(CC) $(CFLAGS) -o $@ -c $<
31

   clean:
       rm -rf example1-shared example1-static example1.o core *~

   # eof
```

That A_tunable, x_view, and y_view are shared with the library implies you can continue to operate on the data to which these views point as you normally would. For instance, you can call dense BLAS operations, such as a dot products or scalar-vector multiply (the so-called "axpy" operation), on x and y. You might also choose to introduce calls to the OSKI kernels selectively, or gradually over time.

3.5 Linking

The exact procedure for linking depends on your platform, and whether you have chosen to use static libraries or shared libraries. Listing 2 shows a sample `Makefile` in that builds Listing 1 on page 9 on a Linux system.

Take note of a few aspects of this `Makefile`:

- **OSKI installation directories**: Lines 4–6 refer to the various OSKI installation paths, relative to the default root of `/usr/local`; see Section 2.2 on page 5.

- **Site module files** (lines 10 and 12): OSKI is divided into a number of smaller modules, including one module for each of the possible sparse matrix data structures OSKI will choose from when tuning.

  To keep you from having to remember what modules are available in your installation, the installer places two files in `${OSKIDIR}/lib/oski` called `site-modules-shared.txt` and `site-modules-static.txt` which list the names of the available shared libraries and static libraries, respectively, as link flags. These files can be pasted right onto the command-line as done in this example.

- **Setting the run-time link path**: In this example, `-Wl,-rpath -Wl,$(OSKILIBS_DIR)` sets this executable's run-time path to point to the directory containing the OSKI library. This path can also usually be specified when the program runs by including it in an environment variable like `LD_LIBRARY_PATH` on Linux, or `LIBPATH` on AIX.

- **Including whole modules during static linking**: The `-Wl,--whole-archive` flag asks the linker to include all of the objects from the list of libraries that follow. If linking against OSKI statically, this flag is important to ensure all OSKI symbols are available at run-time. The matching `-Wl,--no-whole-archive` flag turns off the whole-archive inclusion for subsequent libraries listed on the link command-line.

If you have difficulties or questions, please post them at the help forums on the OSKI home page.

4 Overview of Tuning by Example

The example of Section 3 on page 8 shows the basics of calling OSKI but does not perform any tuning. There are two primary tuning styles in OSKI, summarized as follows:

1. **Tuning using explicit workload and structural hints** (Section 4.1): Any information you can provide *a priori* is information the library in principle does not need to rediscover, thereby reducing the potential overhead of tuning. In this tuning style, you provide the library with structured hints that 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). You then call a "tune routine" which uses this information to choose a new data structure whose performance is optimized for the specified workload.

2. **Tuning using an implicit workload** (Section 4.2 on page 14): The library needs to know the anticipated workload to decide when the overhead of tuning can be amortized, but you may not always be able to estimate this workload before execution. Rather than specifying the workload explicitly, you may rely on the library to monitor kernel calls to determine the workload dynamically. You still explicitly call the same tune routine as above to perform optimizations, but this routine optimizes based on an inferred workload.

To get a better sense of which style is most appropriate for your application, see the examples below.

4.1 Tuning style 1: Providing explicit hints

In this style, you tune a sparse matrix object by optionally providing one or more "hints," followed by an explicit call to the matrix tuning routine, **oski_TuneMat**. Hints describe

Listing 3: **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 );

/* \ldots */
{
    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 );
            // ...
        }
        // ...
    }
}
```

the expected workload, or assert performance-relevant structural properties of the matrix non-zeros.

Listing 3 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, you 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 [14]. We describe a variety of other hints in Section 5.4 on page 22. These hints are directly related to candidate optimizations explored in our research, 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 platform-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, you may call the tuning hints as often as and whenever you choose. For example, suppose the user mixes calls to SpMV and $A^TA \cdot x$ in roughly equal proportion. You specify such a workload as follows:

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

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.

4.2 Tuning style 2: Implicit profiling

Sparse matrix objects may also be tuned without any explicit hints. In this case, OSKI quietly monitors 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 4 on the following page. 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.

5 Guide to the Complete Interface

The available library routines fall into 6 broad categories, summarized as follows:

1. **Initializing OSKI** : You must call **oski_Init**() as shown in Listing 1 on page 9 before calling other routines in the library.

Listing 4: **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 ...
}
```

2. **Creating and modifying sparse matrix and dense vector objects** (Section 5.2 on the following page; Table 1 on page 17): 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 34 defines currently supported input matrix formats.) The user may specify whether the library and the user "share" the input matrix arrays (Section 5.2.1 on the next 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.

3. **Executing kernels** (Section 5.3 on page 19; Table 5 on page 21), *e.g.*, sparse matrix-vector multiply, triangular solve: The interfaces to our kernel routines mimic the "look-and-feel" of the BLAS.

4. **Tuning** (Section 5.4 on page 22; Table 8 on page 23): 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.

5. **Saving and restoring tuning transformations** (Section 5.5 on page 28; Table 14 on page 28): 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 [7]. 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 [12]), for representing such transformations. We provide a high-level overview in the OSKI design document [20]. We expect the details of this language to be continually refined in future releases.

6. **Error-handling** (Section 5.6 on page 29; Table 15 on page 30): 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 35. The following text provides an overview of the semantics and intent behind these bindings.

5.1 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: oski_value_t and oski_index_t. By default, these types are bound to the C double and int types, respectively.

The library build process allows the user to generate 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.

To override the default bindings, the user should include an alternate header file, `oski/oski_T`*xy*`.h`, instead of `oski/oski.h`, where *xy* indicates the desired binding. To bind oski_index_t to int or long, let *x* be `i` or `l`, respectively. Similarly, to bind oski_value_t to float, double, complex-float, or complex-double, let *y* be `s`, `d`, `c`, or `z`, respectively. For example, to use the OSKI bindings with oski_index_t=long and oski_value_t=float, the user would include `oski/oski_Tls.h` instead of `oski/oski.h` in Listing 1 on page 9.

It is possible to mix types within a single source file. See Appendix C on page 60 for details.

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.

5.2 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 on the next page, and C bindings appear in Appendix B.1 on page 35.

5.2.1 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.

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 compressed sparse column (CSC) format 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 35.

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 34.

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>, <k>, <$property\_1$>, ..., <$property\_k$>);

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 page 19), 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 5.2.2 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 9 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 20, 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.

5.2.2 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 35).

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 35 allow, but do

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 5.2.1 on page 16.

not require, the library implementation to report attempts to change implicit zeros to nonzero values as errors.

5.2.3 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 the following page). 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 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 5.4 on page 22.

5.3 Executing kernels

We summarize the available kernels in Table 5 on page 21, and present their bindings in Appendix B.3 on page 47. 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 [21, 22].

*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 5.2.3 on page 19). See Appendix B.3 on page 47 for bindings.

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.

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

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 $A^H A$.

5.3.2 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.

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.

5.3.3 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 $k \geq 2$.

5.3.4 Compatible dimensions for matrix multiplication

All of the kernels apply a matrix $\text{op}(A)$ to a (multi)vector x and store the result in another (multi)vector y. Let $m \times n$ be the dimensions of $\text{op}(A)$, 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 $k = l$.

5.3.5 Floating point exceptions

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

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

- **Workload specification** (Section 5.4.1 on the next page): 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 5.4.2 on the following page): 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 5.4.3 on page 25): 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 4 on page 12 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 [13, 19, 4, 10, 5] or to create dense block structure [15]. 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 5.4.4 on page 25 discusses functionality that allows the user, if she

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

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 52.

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.

5.4.1 Providing workload hints explicitly

Each of the kernels listed in Table 5 on page 21 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) 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 on page 25). 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 on page 25). 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.

5.4.2 Providing structural hints

A user provides one or more structural hints by calling **oski_SetHint** as illustrated in Sections 4.1–4.2. 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

	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 23), the user may pass in one of the above symbolic views.

dominated by dense blocks of a particular size. Rather than just indicate the presence of a single block size by the following call

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 the previous page 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.

5.4.3 Initiating tuning

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

5.4.4 Accessing the permuted form

The interface defines several routines (Table 13 on the next page) that allow the user to determine whether the library has optimized kernel performance by reordering the rows

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_IsMatPermuted	Determine whether a matrix has been tuned by reordering its rows or columns.
oski_ViewPermutedMat	Returns a read-only matrix object for the reordered 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 56.

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 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 5 on the next 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**.

Listing 5: **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 '\Type{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 Â */

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, Â = P_r · A · 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 ← A^k · x in three steps.
     *     1. y ← P_r · y, x ← P_c · x */
    oski_PermuteVecView( Pr, OP_NORMAL, y_view );
    oski_PermuteVecView( Pc, OP_NORMAL, x_view );

    /*     2. y ← A^k · x */
    for( iter = 0; iter < max_power; iter++ )
        oski_MatMult( A_fast, OP_NORMAL, 1.0, x_view, 1.0, y_view );

    /*     3. y ← P_r^T · y, x ← P_c^T · 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 \Var{Pr}, \Var{Pc}, and \Var{A\_fast} */
oski_DestroyVecView( x_view ); oski_DestroyVecView( y_view );
```

Listing 6: **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 );

/* ... */
```

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 58.

5.5 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). 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 6 and Listing 7 on the following page.

Listing 7: **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, ... );

/* \ldots */
```

5.6 Handling errors

The OSKI interface provides two methods for handling errors:

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 on the next page.

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, ... );
```

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 59.

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 ≠ veclen:

```
507     if ( n ≠ veclen ) {
508         your_handler( ERR_DIM_MISMATCH, "oski_MatMult: Mismatched dimensions",
                "oski_MatMult.c", 507,
510             "Matrix dimensions: %d x %d, Vector length: %d\n", m, n, veclen );
            return ERR_DIM_MISMATCH;
512     }
```

6 Troubleshooting

If you are having difficulty with OSKI, here are a few things to try to help track down the problem.

6.1 Installation problems

The `configure` script generates a log of its execution in `config.log`. Inspecting this file, or including it in problem reports, can help identify configuration problems.

6.2 Run-time errors

When running your application, if OSKI reports errors that you can't otherwise figure out, try setting the environment variable `OSKI_DEBUG_LEVEL` to a value of 1 or higher before running your application:

```
%  env OSKI_DEBUG_LEVEL=10 ./your_application ...
```

This causes the library to print diagnostic messages to standard error. Inspecting this output (or including snippets of it in problem reports) may help identify the problem.

6.3 Tuning difficulties

If OSKI does not seem to be tuning, first try enabling run-time debug messages at level 10 (see Section 6.2). OSKI's performance tuning heuristics will print data that helps explain why it failed to tune. Keep in mind that OSKI does a cost-benefit analysis to determine whether or not to tune, and a large number of kernel calls may be required to trigger tuning.

You can also try running any of the pre-built OSKI benchmarks on your matrix and a synthetic workload; see Section 6.4 on the following page.

You may also wish to inspect the off-line benchmarking data collected during installation to get an idea of what performance improvements you might expect. These files all have `.dat` extensions, and may be found in `${OSKIDIR}/lib/oski/bench`. These files are organized by data structure (*e.g.*, CSR, or its blocked variant, block compressed sparse row (BCSR) format) and by type (*e.g.*, `int-double`), and show performance in Mflop/s for some kernel on some test matrix as a function of the data structure parameters.

For instance, the file

```
${OSKIDIR}/lib/oski/bench/profile_MBCSR_MatMult_Tid.dat
```

contains SpMV data for a dense matrix stored in sparse format for the modified block compressed sparse row (MBCSR) format data structure with the `int-double` type combination. Lines in this file might look like

```
1 1 1 273.755 5.03946 % return MBCSR(InputMat, 1, 1)
1 2 1 347.543 8.3001 % return MBCSR(InputMat, 1, 2)
1 3 1 386.285 8.4383 % return MBCSR(InputMat, 1, 3)
1 4 1 401.891 8.7179 % return MBCSR(InputMat, 1, 4)
...
```

In this case, the first 2 columns are the row and column block size, and the fourth column shows the corresponding performance in Mflop/s, and the text after the '%' symbol show the OSKI-Lua transformation program describing that data structure. If you know your matrix has lots of dense block substructure, look at factors of the structural block size in this file. Comparing these data to the data in the corresponding files for CSR or CSC will give some indication as to whether or not you should even expect significant speedups.

6.4 Pre-built synthetic benchmarks

The installation process installs several binary executables in `${OSKIDIR}/bin` that call OSKI on simulated matrix workloads. These programs are listed below, where the suffix `_T`*xy* is a placeholder for the specific integer/floating-point type combinations you selected during installation, where *x* is either `i` for `int` or `l` for `long`, and *y* is one of `s`, `d`, `c`, `z` for single-precision real, double-precision real, single-precision complex, and double-precision complex, respectively.

- `oskitimer`

  The OSKI timer measures time in arbitrary units of 'ticks', which are usually (but not always) equivalent to cycles. You can use this utility to print the tick-to-seconds conversion factor, which will generally match the clock rate of your machine.

- `oskibench_T`*xy*

  Benchmarks OSKI on a user-specified workload and sparse matrix, using a particular user-specified OSKI-Lua tuning transformation.

- `oskibench_autotune_T`*xy*:

  This utility is the same as `oskibench_T`*xy* except OSKI selects the OSKI-Lua transformation based on the workload. Thus, you can use it to test OSKI's automatic tuning.

Run these utilities with no options to see their usage.

Acknowledgements

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] E. CUTHILL AND J. MCKEE, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the ACM National Conference, 1969.

[6] 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.

[7] 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.

[8] ——, *FFTW 3 home page*, 2005. `fftw.org`.

[9] K. GOTO, *High Performance BLAS*, 2005. `www.cs.utexas.edu/users/flame/goto`.

[10] G. HEBER, R. BISWAS, AND G. R. RAO, *Self-avoiding walks over adaptive unstructured grids*, Concurrency: Practice and Experience, 12 (2000), pp. 85–109.

[11] M. A. HEROUX, R. A. BARTLETT, V. E. HOWLE, R. J. HOEKSTRA, J. J. HU, T. G. KOLDA, R. B. LEHOUCQ, K. R. LONG, R. P. PAWLOWSKI, E. T. PHIPPS, A. G. SALINGER, H. K. THORNQUIST, R. S. TUMINARO, J. M. WILLENBRING, A. WILLIAMS, AND K. S. STANLEY, *An Overview of the Trilinos Project*, ACM Transactions on Mathematical Software, 31 (2005), pp. 397–423.

[12] 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`.

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

[20] R. VUDUC, J. W. DEMMEL, AND K. A. YELICK, *An interface for a self-optimizing sparse matrix kernel library*, 2005. `bebop.cs.berkeley.edu/oski`.

[21] R. VUDUC, A. GYULASSY, J. W. DEMMEL, AND K. A. YELICK, *Memory hierarchy optimizations and bounds for sparse* $A^T Ax$, 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.

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

[23] R. C. WHALEY, *Automatically Tuned Linear Algebra Software (ATLAS) home page*, 2005. `netlib.org/atlas`.

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 $\{k : \text{Aptr}[i - 1] \leq k < \text{Aptr}[i] \text{ and } \text{Aind}[k] = j - 1\}$. Then $A(i, j) = \sum_{k \in K} \text{Aval}[k]$. (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 $\{k : \text{Aptr}[i-1] \leq k < \text{Aptr}[i] \text{ and } \text{Aind}[k] = j\}$. Then $A(i, j) = \sum_{k \in K} \text{Aval}[k]$. (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 $\{k : \text{Aptr}[j - 1] \leq k < \text{Aptr}[j] \text{ and } \text{Aind}[k] = i - 1\}$. Then $A(i, j) = \sum_{k \in K} \text{Aval}[k]$. (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 $\{k : \text{Aptr}[j-1] \leq k < \text{Aptr}[j] \text{ and Aind}[k] = i\}$. Then $A(i,j) = \sum_{k \in K} \text{Aval}[k]$. (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 5.6 on page 29, 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 5.6. 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 34.

mode [input] See Table 2 on page 19.
Specifies the copy mode for the arrays **Aptr**, **Aind**, and **Aval**. See Section 5.2.1 on page 16 for a detailed explanation.

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

property_1, . . . **property_k** [input; optional] See Table 3 on page 20.
The user may assert that the input matrix satisfies zero or more properties listed in Table 3 on page 20. 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 20) [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 9.

---

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 $\times$ **num_cols** [input] **num_rows** ≥ 0, **num_cols** ≥ 0
Dimensions of the input matrix.

Aptr, **Aind**, **Aval** [input] **Aptr**, **Aind**, **Aval** $\neq$NULL
The input matrix pattern and values must correspond to a valid CSC representation, as defined in Appendix A on page 34.

mode [input] See Table 2 on page 19.
Specifies the copy mode for the arrays **Aptr**, **Aind**, and **Aval**. See Section 5.2.1 on page 16 for a detailed explanation.

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

property_1, ... **property_k** [input; optional] See Table 3 on page 20.
The user may assert that the input matrix satisfies zero or more properties listed in Table 3 on page 20. 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 20) [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 9, 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] $1 \leq$ **row** $\leq m$, $1 \leq$ **col** $\leq n$
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** $\geq$1, **num_cols** $\geq$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 5.2.3 on page 19) 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**×**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 9, 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** $\geq$1, **num_cols** $\geq$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 5.2.3 on page 19) 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 ( 1    .125 ).
//                    ( .125  1   ).
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, ( 1   −2 ).
//                                   ( −2   1 ).
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 ( 1    .125 )
//                            ( .125  1   )
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] $1-\text{m} \leq$**diag_num** $\leq \text{n}-1$
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] — $1-\text{m} \leq$ **diag_num** $\leq \text{n}-1$
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_IMPLICIT.

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 9, 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 9.

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 9.

---

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 $X = \left(x_1 \cdots x_k\right)$, 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_LEADDIM].

Example:
// Let A be the matrix shown in Listing 1 on page 9, 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 9, 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 9 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 $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$, where $\text{op}(A) \in \{A, A^T, A^H\}$.

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

opA [input] See Table 6 on page 21.
Specifies $\text{op}(A)$.

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 9.

---

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 21.
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 9. 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 21.
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 9, and let $x^T = \begin{pmatrix} .1 & .2 & .3 \end{pmatrix}$. The following code computes

```
// t ← A · x, and y ← A^T A · 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 ← A· x, y ← A^T A· 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 $z \leftarrow \omega \cdot \mathrm{op}(A) \cdot x + \zeta \cdot z$, where $\mathrm{op}(A) \in \{A, A^T, A^H\}$.

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 21.
Specifies $\mathrm{op}(A)$.

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 $\mathrm{op}(A)$, w, and z have compatible dimensions, then this routine computes $y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$ and $z \leftarrow \omega \cdot \mathrm{op}(A) \cdot w + \zeta \cdot z$ 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 9, and let x^T = (.1  .2  .3). This example computes
//       y ← A·x and z ← A^T·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 $y \leftarrow \alpha \cdot \text{op}(A)^{\rho} \cdot x + \beta \cdot y$. Also optionally computes $T = \begin{pmatrix} t_1 & \cdots & t_{\rho-1} \end{pmatrix}$, where $t_k \leftarrow \text{op}(A)^{k} \cdot x$ 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 21.
Specifies $\text{op}(A)$.

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 = (t1 t2).
// Initially, let t1^T = (.1 .1 .1), and t2^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 t1, t2.
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 = A^2*x = [ 0.0625 ; 0.5625 ; 2.375 ];"
//      "y = A^3*x = [ 0.015625 ; 0.421875 ; 2.5625 ];"
printf( "t1 = A*x = [ %f ; %f ; %f ];\n", T[0], T[1], T[2] );
printf( "t2 = A^2*x = [ %f ; %f ; %f ];\n", T[3], T[4], T[5] );
printf( "y = A^3*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 $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$, 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.

opA [input] See Table 6 on page 21.
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 25).
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 25).
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 3 on page 13.

---

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 21.
Specifies op(T).

alpha [input]
Scalar constant α.

x_view [input] **x_view** is valid or symbolic (see Table 11 on page 25).
View object for a (multi)vector x.

num_calls [input] **num_calls** is non-negative or symbolic (see Table 10 on page 25).
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 21.
Specifies $\text{op}(A)$.

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 25).
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 25).
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 21.
Specifies op(A).

x_view, **y_view**, **w_view**, **z_view** [input] Vectors are valid or symbolic (see Table 11 on page 25).
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 25).
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 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 $T = \begin{pmatrix} t_1 & \cdots & t_{\rho-1} \end{pmatrix}$, 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 21.
Specifies 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, **y_view** [input] Vectors are valid or symbolic (see Table 11 on page 25) 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 $T = \begin{pmatrix} t_1 & \cdots & t_{\rho-1} \end{pmatrix}$, or SYMBOLIC_MULTIVEC.

num_calls [input] **num_calls** is non-negative or symbolic (see Table 10 on page 25).
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 24.
User-specified structural hint. This hint may be followed by optional arguments, as listed and typed in Table 9 on page 24.

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 3 on page 13.

---

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 26, or an error code otherwise.

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

Example:
See Listing 3 on page 13 and Listing 4 on page 15.

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 5 on page 27.

---

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 5 on page 27.

---

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 5 on page 27.

---

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 5 on page 27.

---

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 5 on page 27.

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 6 on page 28.

---

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 7 on page 29.

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 Mixing Types

The OSKI implementation supports "mixing" of basic data types (Section 5.1 on page 16) within a source file by providing type-specific subroutine bindings, which a user may call explicitly. OSKI uses the C preprocessor to generate these bindings, and the user can access their prototypes as shown below.

For example, suppose a user needs bindings for both one type of matrix which uses int for the indices and float for the values (*i.e.*, a `Tis` matrix), and another which uses long for the indices and double for the values (`Tld`). She should first include the type-specific headers as follows:

```
#include <oski/oski_Tis.h> /* (int, float) */

#define OSKI_REBIND /* Reset bindings */
#include <oski/oski_Tld.h> /* (long, double) */
```

The first include directive uses the C preprocessor to "bind" the standard OSKI subroutine names to "mangled" names that include the type information. The effect is to introduce a

define directive of the form shown below, along with a C prototype for the type-specific name, as if the user had entered the following:

```
#define oski_MatMult oski_MatMult_Tis
int oski_MatMult (int *, int *, float *, ...);
```

The second include directive, preceeded by the macro definition, OSKI_REBIND, first undefines the standard names previously bound to the `Tis` type, and then redefines them to the `Tld` type. The user should insert additional OSKI_REBIND and include directives for other types as necesary.

The header files declare all the type-specific prototypes, and the type-mangled names are now available for the user to call explicitly. Note that the basic types (*e.g.*, oski_matrix_t), are similarly available in type-mangled forms (*e.g.*, oski_matrix_t_Tis, oski_matrix_t_Tld).

```
/* Matrix A, in (int, float) CSR format */
int *Aptr, *Aind;  float *Aval;
/* Matrix B, in (long, double) CSR format */
long *Bptr, *Bind;  double *Bval;
/* ... */
oski_matrix_t_Tis A = oski_CreateMatCSR_Tis (Aptr, Aind, Aval, ...);
oski_matrix_t_Tld B = oski_CreateMatCSR_Tld (Bptr, Bind, Bval, ...);
/* ... */
oski_MatMult_Tis (A, ...);
oski_MatMult_Tld (B, ...);
/* ... */
```

D OSKI Library Integration Notes

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

D.1 Sparse BLAS

The recent revision of the BLAS standard defines an interface for sparse matrix kernels [6, 3]. Indeed, the very first OSKI design proposed to add just a single routine to the Sparse BLAS for explicit tuning (analogous to **oski_TuneMat**) and a few additional routines to provide the new cache-friendly kernels like sparse $A^T A \cdot x$ and $A^k \cdot x$ [22, Chapter 8]. Although the current OSKI design departs from the Sparse BLAS in several ways, the similarities make OSKI a suitable basis for building a Sparse BLAS implementation. Below, we describe the main differences and outline what a Sparse BLAS implementation based on OSKI might look like.

The primary ways in which OSKI differs from the Sparse BLAS interface are as follows:

- **Explicit tuning**: In OSKI, the user must call a tuning routine explicitly, thereby exposing the point during program execution at which the non-trivial cost of tuning may occur. In the Sparse BLAS, any such tuning happens implicitly.
- **Workload hints for tuning**: Both the Sparse BLAS and OSKI define allow the user to provide hints about matrix structure, for example, dense rectangular block substructure. OSKI also allows *workload* hints, such as the frequency and mix of which kernels will be called, so that an OSKI implementation can perform a cost-benefit analysis to decide if tuning will be profitable given its run-time cost. However, workload hints are optional; if not given, OSKI will try to infer the workload implicitly based on

what kernels the user calls. An implicit profiling style is the only mode available in the Sparse BLAS.

- **Additional kernels**: In addition to level-2 and level-3 versions of matrix-vector multiply and triangular solve, OSKI includes kernels that have more inherent reuse of the matrix: simultaneous multiplication by A and A^T, sparse $A^T A \cdot x$, and matrix-powers multiplication $A^k \cdot x$. If these kernels prove useful, they should be considered in a future revision of the Sparse BLAS interface.

- **Explicit permutation representation**: One possible optimization is to permute the rows and columns of the matrix to improve temporal or spatial locality, and the user can access these permutations explicitly through the OSKI interface if her algorithm can amortize their cost (see Section 5.4.4 on page 25).

- **Tuning save/restore mechanism**: OSKI provides a way to determine what tuning transformation was applied and also to request a specific transformation Section 5.5 on page 28.

The following outline shows how a Sparse BLAS could be implemented using OSKI by mapping the various components of OSKI, described in Sections 5.1–5.6, to the Sparse BLAS interface.

1. Basic scalar types (Section 5.1 on page 16): OSKI supports the same basic floating point types (single, double, and their complex variants) as the BLAS.

2. Matrix handle creation (Section 5.2 on page 16): The Sparse BLAS routine which ends the matrix assembly should construct a CSR or CSC matrix internally, and call the OSKI matrix handle creation routine on this matrix. The Sparse BLAS implementation needs to maintain a mapping between the handle it returns to the user and the OSKI handle. Since the matrix is logically fixed in the Sparse BLAS once assembly ends, the OSKI routines to get/change non-zero values are irrelevant.

3. Kernels (Section 5.3 on page 19: OSKI does not define any sparse or dense vector kernels. The sparse matrix-dense vector/matrix multiply and triangular solve kernels correspond directly to the equivalent OSKI routines. The Sparse BLAS kernels just need to wrap the dense vector/matrix data by creating vector views, which are essentially just semantic wrappers around their data (see Section 5.2 on page 16).

   Higher-level kernels like $A^T A \cdot x$ are not available in the Sparse BLAS and are thus not relevant to an OSKI-based implementation.

4. Tuning (Sections 5.4–5.5): The simplest way to enable tuning is to use the implicit profiling style supported by OSKI (Section 4.2 on page 14): at the end of every SpMV or sparse triangular solve (SpTS) call, the Sparse BLAS calls **oski_TuneMat**. In addition, any Sparse BLAS structural hints provided by the user can be mapped directly to equivalent OSKI structural hints and provided to OSKI during matrix assembly.

   Another possible style is to assume an arbitrarily large workload (*e.g.*, always assume 100 SpMV operations, or use ALWAYS_TUNE_AGGRESSIVELY), and specify this information using a workload hint at matrix assembly time. Then, the Sparse BLAS implementation would call **oski_TuneMat** before returning from the end-assembly routine, thus hiding the cost of tuning in the assembly cost. This style is the "explicit tuning" style described in Section 4.1 on page 12.

A third more explicit tuning style is possible. Any OSKI implementation will make its data structure tuning transformations available via the tuning save/restore interface. Thus, all the tuning logic could be implemented internally in the Sparse BLAS, using OSKI only to implement a particular transformation.

5. Error handling (Section 5.6 on page 29): OSKI provides both error return codes and a way to call a user-defined error handler. Since the Sparse BLAS uses only error code returns to communicate errors, an implementation would simply disable the error handler by passing **oski_SetErrorHandler** a NULL function argument.

D.2 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 [2, 1]. We are currently implementing such an extension, and a "pre-alpha" implementation is available from the OSKI home page. Comments and contributions are welcome.

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 34): 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 5.2.1 on page 16) 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 4.2 on page 14, 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.

  The current "pre-alpha" implementation uses the explicit style, tuning at the end of matrix assembly, and also upon any explicit conversion.

- 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 20).
- 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.

D.3 MATLAB*P

MATLAB*P is a research prototype system providing distributed parallel extensions to MATLAB [18]. 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 4.2 on page 14.

D.4 Kokkos (Trilinos)

The Trilinos solver package is another possible target for OSKI integration [11]. Trilinos provides a uniform C++ interface for its users and solver components through Kokkos, the core BLAS-like sparse and dense kernel package. OSKI could be used to implement the abstract interfaces defined in Kokkos to complement Kokkos' existing sparse format implementations.