pOSKI: Parallel Optimized Sparse Kernel Interface Library
User's Guide for Version 1.0.0

Jong-Ho Byun *Richard Lin*
James W. Demmel *Katherine A. Yelick*

Berkeley Benchmarking and Optimization (BeBOP) Group
University of California, Berkeley
http://bebop.cs.berkeley.edu/poski

April 27, 2012

Note: Some of this document is copied from the OSKI User's Guide for Version 1.0.1h written by Richard Vuduc, James Demmel and Katherine Yelick in 2007.

Contents

1 What is pOSKI and who should use it?

The parallel Optimized Sparse Kernel Interface (pOSKI) [11] is a collection of kernels that provide automatically tuned ("autotuned") high performance computational kernels for sparse matrices, such as Sparse-Matrix-Vector-Multiplication (SpMV). pOSKI targets both uniprocessor and multicore machines. pOSKI builds on prior work on OSKI [7,8,10], which provided autotuned kernels for SpMV and other kernels on cache-based superscalar uniprocessors. The purpose of both pOSKI and OSKI is to make it easy for developers of solver libraries, and of scientific and engineering applications, to more easily attain high performance in commonly used sparse matrix operations, via autotuning. Autotuning is done both at installation time (also called off-line tuning), ie. when pOSKI is installed on a particular architecture, which allows extensive benchmarking of different kernel implementations to identify the fastest ones, and at run-time, when more is known about the user's matrix, and the best data structure and kernel implementation can be selected quickly. pOSKI also lets the user cheaply reuse a prior tuned data structure and implementation, exploiting the fact that the same matrix structure is often reused.

pOSKI is a part of the on-going work by the Berkeley Benchmarking and Optimization (BeBOP) group [10], a research program on automatic performance tuning and analysis at the University of California, Berkeley.

1.1 What optimizations does pOSKI perform?

The primary aim of pOSKI is to provide parallel functionality, which includes additional optimizations presented in previous work done [1, 5, 6] for sparse matrix computations, based on the OSKI library. The optimizations include register blocking, thread blocking, software prefetching, software pipelining, SIMD, and loop unrolling.

Automatic tuning in pOSKI happens in two phases: (a) *off-line tuning*, which is run automatically during pOSKI installation on a particular machine, and (b) *run-time tuning*, when the user calls pOSKI's tuning routines with a particular sparse matrix.

For parallel functionality, pOSKI supports several parallel programming models (called *threading models)* to create multiple threads on multicore architectures, and it also supports several *partitioning models* to split a matrix into sub-matrices. For more detail on supported threading models and partitioning models in pOSKI, see Section 4.

Note: This version of pOSKI supports only limited functionalities of OSKI library. For more detail for user's interface, see Section 4.

1.2 How to read this documentation

The fastest way to get started is to look at the material in Sections 2 and 3, which contains enough information to build and install pOSKI and to write the first test program. If the user runs into problems, see Section 5 for debugging hints, or post a question in the online forum through the pOSKI home page [11] (bebop.cs.berkeley.edu/poski).

2 Installation

This section describes how to compile and install pOSKI-1.0.0 from its source. Automatic tuning in pOSKI happens in two phases: *off-line tuning*, which is run automatically during pOSKI installation on a particular machine, and (b) *run-time tuning*, when the user calls pOSKI's tuning routines with a particular sparse matrix.
Off-line tuning automatically performs the following operations: (1) generates code for various implementations of the register block size, (2) collects the benchmark data of these implementations on user's machine, and (3) selects the best implementation for each register block size. These steps are automated and should not require any manual intervention on user's part.

Currently, pOSKI has been tested on the following environments:

- Architectures: Intel Core2 Duo processor (E8400), Intel Nehalem processor (x5550), Intel Core i5-3550, Intel Core i7-3960X
- OS: Linux with kernel-2.6
- C Compilers: gcc-4.3, icc-11.1 with SSE3 or SSE4 supports
- Python 2.7.2+

2.1 What the user will need to get started

There are two ways to install pOSKI library on the user's system: If OSKI-1.0.1h, the last version of OSKI, is already installed, (1) the user can install pOSKI on top of the OSKI library (see Section 2.2), otherwise (2) the user can install pOSKI as a stand-alone library (see Section 2.3).

The user will also need to think about the following issues:

- ***In which directory will pOSKI be installed?***
  pOSKI installs itself in the subdirectories of */usr/local/* by default, but the user can override this. In all of our examples, we will use ${*POSKIDIR*} as a placeholder for the directory of choice. For more details, see Section 2.2 or 2.3.
- ***Which integer and floating-point precisions are needed?***
  Sparse matrices are stored using both scalar integer indices and floating-point values. pOSKI currently supports only the "*int-double"* data type: the C language *int* for the integer types and *double* for the floating-point types. If the user builds pOSKI on top of OSKI, the user must make sure that the OSKI library was built using the "*int-double*" data type. By default, the OSKI installation process builds "*int-double"* data type. For more details about the data types, see the OSKI-1.0.1h User's Guide Document [8].
- ***Does the user want to build static or shared libraries?***
  Currently, pOSKI builds only a shared library.

2.2 How to install pOSKI on top of the installed OSKI library

Follow these steps to compile and install pOSKI on top of installed OSKI. We show sample command-lines for *sh/csh*-compatible shells, where *%* denotes the command-line prompt. For more details about the default configuration parameters, see Section 2.4.

For getting started with installing OSKI, see the OSKI-1.0.1h User's Guide Document [8].

(1) **Download** *poski-1.0.0-part.tar.gz* from the pOSKI home page.
(2) **Unpack** the distribution
% gunzip –c poski-1.0.0-part.tar.gz | tar –xvf –
This command unpacks the pOSKI distribution into a subdirectory named *poski-v1.0.0*.
(3) **Set OSKIDIR**
% export OSKIDIR = {directory path to installed directory of OSKI-1.0h}
% export OSKIBUILD = {directory path to built directory of OSKI-1.0h}
(4) **Run the *INSTALL.sh*** script file in *poski-v1.0.0/* directory
% mkdir build-1.0.0
% cd build-1.0.0
% ../poski-v1.0.0/INSTALL.sh --prefix=${POSKIDIR}
As done in this example, we strongly recommend building pOSKI in a directory separate from the source tree in Step 2. Here we use a directory named *build-1.0.0*.
User can run each command-line manually instead of running the *INSTALL.sh* script file.
The *INSTALL.sh* script file includes following actions:
a. Configure pOSKI for user's platform:
% ../poski-v1.0.0/configure --prefix=${POSKIDIR}
This step configures *poski-v1.0.0* with optional *prefix*. User may set a particular directory path for pOSKI installation.
b. Compile pOSKI:
% make
This step compiles the source codes.
c. Off-line tuning:
% make benchmarks
This step runs "*off-line tuning*". This step is optional with "*--with-tune=yes*" installation option.
d. Test pOSKI:
% make check
This step runs user's build of pOSKI through an extensive series of tests. This step is optional with "*--with-check=yes*" installation option.
e. Install pOSKI:
% make install
This step installs all of the pOSKI files and benchmarking data into the directory, *${POSKIDIR}*, specified in the first step *a*.

NOTE: Running times of each step can vary widely across platforms, taking anywhere between a few minutes to several hours.

2.3 How to install pOSKI as a stand-alone library

Follow these steps to compile and install pOSKI as a stand-alone library. We show sample command-lines for *sh/csh*-compatible shells, where *%* denotes the command-line prompt. For more details about the default configuration parameters, see Section 2.4.

(1) **Download** *poski-1.0.0-full.tar.gz* from the pOSKI home page.
(2) **Unpack** the distribution

% gunzip –c poski-1.0.0-full.tar.gz | tar –xvf –

This command unpacks the pOSKI distribution into a subdirectory named *poski-v.1.0.0*.

(3) **Run *INSTALL.sh*** script file in *poski-v1.0.0/* directory

% mkdir build-1.0.0

% cd build-1.0.0

% ../poski-v1.0.0/INSTALL.sh --with-oski=yes --prefix=${POSKIDIR}

See step 4 in Section 2.2 for more details.

NOTE: Running times of each step can vary widely across platforms, taking anywhere between a few minutes to several hours.

2.4 Customizing pOSKI 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:

% ../poski-v1.0.0/INSTALL.sh --help

Some default variables of configuration:

(1) Installation directories

pOSKI directory:	*POSKIDIR=/usr/local/*
OSKI directory:	*OSKIDIR=${POSKIDIR}/build_oski/*

(2) Compiler and flags

C compiler:	*CC=gcc*
C compiler flags:	*CFLAGS= –g –O3 –std=gnu99*

(3) Data Type

Indices:	*POSKIINT=int*
Values:	*POSKIVAL=double*

2.4.1 Overriding the default compiler and/or flags

To specify the compiler and /or compiler flags, define the CC and/or CFLAGS environment variables accordingly. For example, to use the Intel C compiler to build pOSKI, user can set the configuration option of the *INSTALL.sh* script file as:

% ../poski-v1.0.0/INSTALL.sh --cc=icc --prefix=${POSKIDIR}

2.4.2 Selecting other or additional scalar type precisions

Currently, pOSKI only uses the C compiler's *int* data type to store integer indices, and *double* for floating-point values. However, if the user wants to know what other data types OSKI supports, see the OSKI-1.0.1h User's Guide Document [8].

3 Using pOSKI: A First Example

This section introduces the C version of pOSKI by an example. The interface uses an object-oriented calling style, where the four main object types are (1) *thread object*, (2) *partition-matrix/vector object*, (3) *tunable-matrix object*, and (4) *vector-view object*. The *thread object* is used to create the multiple threads. The default number of multiple threads (*the number of threads*) equals the number of available cores on the user's system. The *partition-matrix (or vector) object* is used to split a matrix (or vector) into sub-matrices (or sub-vectors). The default number of sub-matrices (*the number of partitions*) equals the number of threads. The *tunable-matrix object* is used for a sparse matrix. The *vector-view object* is used for a dense vector. For more detail, see Section 4.

In addition to showing pOSKI's basic usage and providing the user with a first example to test the pOSKI installation, this example illustrates how the user can gradually migrate an existing application to use pOSKI, provided that the application uses "standard" array representations of sparse matrices in CSR format and dense vectors.

Example 3.1 shows 7 steps to use basic routines in pOSKI library to perform sparse matrix-vector multiplication (SpMV), and Example 3.2 shows a sample *Makefile* for Example 3.1. Calls to pOSKI routines are shown in **blue bold-face**, and pOSKI's objects are shown in **bold-face**. The Example 3.1 shows the basics of calling pOSKI but does not perform any *run-time tuning*. To learn about *run-time tuning* see Section 4.6.

For more detail on the user interface in the pOSKI library, see Section 4.

Example 3.1: This example illustrates basic object creation and kernel execution in pOSKI. Here, we perform one sparse matrix-vector multiply for a sample sparse matrix A using the default thread and partition objects.

```
/* This example computes SpMV ( y = α•Ax + β•y ) with default threading model and partitioning model. */
1.  #include <stdio.h>
2.  #include <poski.h>
3.
4.  int main(int argc, char **argv)
5.  {
6.  STEP 1:  /* Initialize pOSKI library */
7.      poski_Init();
8.
9.  STEP 2:  /* Initialize Sparse matrix A in CSR format, and dense vectors */
10.     int nrows=3;   int ncols=3;  int nnz=5;
11.     int Aptr[4]={0, 1, 3, 5};  int Aind[5]={0, 0, 1, 0, 2};  double Aval[5]={1, -2, 1, 0.5, 1};
12.     double x[3]={.25, .45, .65};   double y[3]={1, 1, 1};
13.     double alpha = -1, beta = 1;
14.
15. STEP 3: /* Create a default thread object {with  #threads = #available_cores} */
16.     poski_threadarg_t   *poski_thread = poski_InitThreads();
17.
18. STEP 4: /* Create a tunable-matrix object by wrapping the partitioned sub-matrices using a thread object and a
    default partition-matrix object {with  #partitions = #threads} */
19.     poski_mat_t    A_tunable =
20.         poski_CreateMatCSR ( Aptr, Aind, Aval, nrows, ncols, nnz,/* Sparse matrix A in CSR format */
21.                              SHARE_INPUTMAT,                    /* <matrix copy mode> */
22.                              poski_thread,                      /* <thread object> */
23.                              NULL,                              /* <partition-matrix object> (NULL: default) */
24.                             2, INDEX_ZERO_BASED, MAT_GENERAL);/* specify how to interpret non-zero pattern */
```

```
25.
26. STEP 5: /* Create wrappers around the dense vectors with  <partition-vector object> (NULL: default) */
27.       poski_vec_t    x_view = poski_CreateVec(x, 3, STRIDE_UNIT, NULL);
28.       poski_vec_t    y_view = poski_CreateVec(y, 3, STRIDE_UNIT, NULL);
29.
30. STEP 6: /* Partition input/output vectors and Perform matrix vector multiply (SpMV),  y = α•Ax + β•y */
31.       poski_MatMult(A_tunable, OP_NORMAL, alpha, x_view, beta, y_view);
32.
33. STEP 7: /* Clean-up interface objects and threads, and shut down pOSKI library  */
34.       poski_DestroyMat(A_tunable); poski_DestroyVec(x_view);  poski_DestroyVec(y_view);
35.       poski_DestroyThreads(poski_thread);
36.       poski_Close();
37.
38.       return 0;
39. }
```

3.1 Initialize pOSKI

To initialize the library, the user's application should include *poski.h* (line 2) and call *poski_Init* (line 7). To release resources, the user should call *poski_Close* (line 36) at the end of the program to shut down the library. For more detail, see Section 4.1.

3.2 Initialize a sample matrix and vectors

The user needs to create arrays of a sparse matrix in CSR format (here, *Aptr, Aind, Aval, nrows, ncols*, and *nnz*) and arrays of dense vectors (here *x* and *y*) (lines 10-12). For more detail, see Section 4.4.

3.3 Create a default thread object

We create and initialize a thread object, *poski_thread*, of type *poski_threadarg_t* by a call to *poski_InitThreads* (line 16). By using the default thread object, this routine creates reusable multiple threads as *threadpool*. The number of threads equals the number of available cores on the system. This step automatically checks the number of available cores on the system. For more detail, see Section 4.2.

3.4 Create a tunable-matrix object

We create a tunable-matrix object, *A_tunable*, of type *poski_mat_t* from the input sparse matrix in CSR format by a call to *poski_CreateMatCSR* (lines 19-24) with the following arguments:

(1) Arguments 1-3 (line 20) specify the arrays of a sparse matrix in CSR format.
(2) Arguments 4-5 (line 20) specify the sparse matrix dimensions.
(3) Argument 6 (line 20) specifies the number of non-zeros of the sparse matrix.
(4) Argument 7 (line 21) 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 *poski_DestroyMat* (line 34), 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 user may choose to free the arrays at any time. We discuss the semantics of both modes in detail in Section 4.4.

(5) Argument 8 (line 22) specifies the thread object of type *poski_threadarg_t*. The given thread object is copied into the tunable-matrix object. As a default, the number of threads is the same as the number of available cores on the system. For more detail on how to modify the default thread object, see Section 4.2.

(6) Argument 9 (line 23) specifies the partition-matrix object of type *poski_partitionarg_t*. The value *NULL* indicates to use the default partition object. In this example, the default partition object is copied into the tunable-matrix object. For more detail on how to modify the default partition object, see Section 4.3.

(7) Arguments 10-12 (line 24) tell the library how to interpret the arrays of the input sparse matrix in CSR format. Argument 10 is a count that says the next 2 arguments are semantic properties needed to interpret the input sparse 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 value *MAT_GENERAL* indicates that the input matrix specifies all non-zeros. For more detail on matrix properties, see Section 4.4.

In this example, think of *A_tunable* as a wrapper around these arrays. Since this example uses the *SHARE_INPUTMAT* copy mode and performs no tuning, pOSKI will not create any copies of the input matrix. However, additional row-index arrays for sub-matrices are created when partitioning a matrix into sub-matrices.

3.5 Create vector-view objects

Dense vector objects, *x_view* and *y_view*, of type *poski_vec_t*, are always wrappers around user array representations (lines 27-28). We refer to such wrappers as views. A vector-view encapsulates basic information about an array, such as its length or the stride between consecutive elements of the vector within the array. The fourth argument is for specifying partition-vector object. In this example with *NULL* argument as a default set, the vector is not specifically partitioned. However, if the fourth argument is specified with a partition object, the vector will be partitioned. For more detail on vector-view objects, see Section 4.5.

3.6 Call a kernel for sparse matrix-vector multiply

The argument lists of kernels, such as *poski_MatMult* for SpMV in this example (line 31), follow the conventions of the BLAS. For example, a user can specify the constant *OP_TRANS* as the second argument to multiply by A^T instead of A, or specify other values for α and β. Here, the constant *OP_NORMAL* indicates to multiply by A. This kernel call performs a check on the partition status of the tunable-matrix and vector-view objects, and then performs an appropriate sparse matrix-vector multiply in parallel. For more detail, see Section 4.7.

3.7 Clean-up and close pOSKI

The calls to *poski_DestroyMat, poski_DestroyVec* and *poski_DestroyThreads* free any memory allocated by the library to these objects (lines 34-35). However, since the user and library share the arrays underlying *A_tunable, x_view,* and *y_view*, user is responsible for freeing these arrays (here, *Aptr, Aind, Aval, x*, and *y*).

The last step is to call *poski_Close* (line 36) at the end of the program to shut down the library.

3.8 Linking

The exact procedure for linking depends on the user's platform. Example 3.2 shows a sample *Makefile* that builds Example 3.1 on a Linux system.

Example 3.2: *Makefile* for the first example. This example is for using pOSKI built on top of OSKI.

```
1.  #Location of installed OSKI library
2.  OSKIDIR = {path_to_oski_build_directory}
3.  OSKIINC = $(OSKIDIR)/include
4.  OSKILIB = $(OSKIDIR)/lib/oski
5.
6.  #Location of pOSKI library
7.  POSKIDIR = {path_to_poski_build_directory}
8.  POSKILIB = $(POSKIDIR)/lib
9.  POSKIINC = $(POSKIDIR)/include/poski
10.
11. #OSKI & pOSKI link flags
12. OSKILIB_SHARED = –I$(OSKIINC) –Wl,–rpath –Wl,$(OSKILIB) –L$(OSKILIB) `cat
    $(OSKILIB)/site-modules-shared.txt` –loski
13. POSKILIB_SHARED = –I$(POSKIINC) –Wl,–rpath –Wl,$(POSKILIB) –L$(POSKILIB) –
    lposki
14. LDFLAGS_SHARED = $(OSKILIB_SHARED) $(POSKILIB_SHARED) –lm
15.
16. CC = icc
17. CFLAGS = –g –O3 –pthread –openmp
18.
19. SRC = example
20. all:$(SRC)-shared
21. $(SRC)-shared: $(SRC).o
22.     $(CC) $(CFLAGS) –o $@ $(SRC).o $(LDFLAGS_SHARED)
23. .c.o:
24.     $(CC) $(CFLAGS) $(LDFLAGS_SHARED) –o $@ –c $<
25. clean:
26.     rm –rf $(SRC)-shared $(SRC).o core*~
```

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

(1) OSKI directories:
Lines 2-4 refer to the various paths of OSKI directories.

(2) pOSKI installation directories:
Lines 7-9 refer to the various pOSKI installation paths.

(3) Setting the run-time link path:
Lines 12-14 refer to the executable run-time path to point to the directory containing the OSKI and pOSKI library. For more details on setting OSKI's site-module-file, see the OSKI-1.0.1h User's Guide Document [8].

If the user has difficulties or questions for testing this first example, please contact us through the pOSKI home page.

4 User Interface

This section illustrates available user interfaces in the pOSKI library. The user interface is divided into 7 broad categories; (1) Initial/Close Handler, (2) Thread Handler, (3) Partition Handler, (4) Matrix Handler, (5) Vector Handler, (6) Tuning Handler, and (7) Kernel Handler.

4.1 Initial/Close Handler

This module includes the routines, shown in Table 4.1, to initialize and shut down the library. The user must call *poski_Init()* as shown in Example 3.1 before calling other routines in the library. To release resources, the user also calls *poski_Close()* at the end of user's program to shut down the library.

Routine	Description
poski_Init	Initialize pOSKI library.
poski_Close	Shut down pOSKI library.

Table 4.1: Initialize/close the library. Bindings appear in Appendix A.1.

4.2 Thread Handler

This module includes the routines, shown in Table 4.2, to create/modify/destroy a thread object of type *poski_threadarg_t*. Currently, the thread object contains information on the <*threading model*>, <*number of threads*>, and <*available cores on system*>.
We define a thread object as global or local. The global thread object can be used multiple times when the user creates multiple tunable-matrix objects. The local thread object is a copy of the global thread object in a specified tunable-matrix object, and it can be used only for the matrix object.

Routine	Description
poski_InitThreads	Create a default thread object from available system cores.
poski_DestroyThreads	Free a thread object.
poski_ThreadHint	Specify hints about the threading model and the number of threads. For a list of available options for the threading model, see Table 4.3.
poski_report_threadmodel	Report information on the currently used threading model.

Table 4.2: Creating, modifying and destroying a thread object. Bindings appear in Appendix A.2.

4.2.1 Creating a thread object

User must call *poski_InitThreads()* as shown in Example 3.1 before calling the routines for partition, matrix, vector, tuning, and kernel handlers. The routine checks the number of available cores on the system, and creates a thread object with the default <*threading model*>, *POSKI_THREADPOOL*. The available threading models are shown in Table 4.3. The *POSKI_THREADPOOL* tells the library to create reusable threads once and may use them multiple times to perform multiple tasks, while other threading models tell the library to create and destroy threads at each task. The default number of

threads is the number of available cores on the system. For more detail on the threading models, see Appendix B.

Hint	Option	Description
<threading model>	*POSKI_THREADPOOL	User and library agree to create reusable threads, *threadpool*, once and to use it to perform multiple tasks.
	POSKI_PTHREAD	User and library agree to create and destroy threads at each task using Pthread.
	POSKI_OPENMP	User and library agree to create and destroy threads at each task using OpenMP.

Table 4.3: Available options for the threading model. The default option is marked by an asterisk (*).

4.2.2 Modifying a thread object

The user may call *poski_ThreadHints()* to modify the thread object. The first and second arguments of this routine are the *<thread object>* and *<tunable-matrix object>* which the user wants to modify. The user must set at least one of these two objects or both. For instance, if the user wants to modify only the local thread object in a specified tunable-matrix object, the user must assign NULL for the first argument *<thread object>* and a valid tunable-matrix object for the second argument *<tunable-matrix object>*. The third argument is *<threading model>* to set the user's desired threading model, and the last argument is *<number of threads>* to set the user's desired number of threads. Moreover, this routine automatically adjusts the number of threads to satisfy the following conditions:

(1) The number of threads must be between 1 and the number of available cores on the system.
(2) If the second argument *<tunable-matrix object>* is given, the number of threads must be equal to (*the number of partitions / k*), where *k* is an integer.

For instance, when applying only the first condition, this routine sets the number of threads equal to the number of available cores on the system if the given *<number of threads>* is greater than the number of available cores on the system. Example 4.1 sketches two simple examples for using hints to modify the thread object.

Example 4.1: This example illustrates basic thread object modification in pOSKI. Here, we create a default thread object and modify it with the user's hint in two different ways; (a) modification of a global thread object, and (b) modification of a local thread object in a specified tunable-matrix object. Here we assume the number of available cores on the system is 16, and the default partition object, *NULL*, is used.

(a) Modification of a global thread object

```
poski_threadarg_t   *poski_thread = poski_InitThreads();
poski_ThreadHints(poski_thread, NULL, POSKI_OPENMP, 8);
poski_mat_t   A_tunable = poski _CreateMatCSR (..., poski_thread, NULL, ... )
```

(b) Modification of a local thread object in a specified tunable-matrix object

```
poski_threadarg_t   *poski_thread = poski_InitThreads();
poski_mat_t   A_tunable = poski _CreateMatCSR (..., poski_thread, NULL, ... )
```

poski_ThreadHints*(NULL, A_tunable, POSKI_OPENMP, 8);*

In Example 4.1(a), first we create a default thread object, *poski_thread*, as a global thread object that is set with *POSKI_THREADPOOL* and 16 threads. Second, we modify only the global thread object, *poski_thread,* with *POSKI_OPENMP* and 8 threads. Here, the given number of threads could be adjusted to satisfy the first condition, although that is not necessary here. Then we create a tunable-matrix object using the global thread object. Thus, *poski_CreateMatCSR* uses *POSKI_OPENMP* and 8 threads, and creates 8 sub-matrices if the default partition object is used. In this case, when user calls kernel routines with the tunable-matrix object *A_tunable*, the kernel routines use *POSKI_OPENMP* and 8 threads for 8 sub-matrices in parallel.

In Example 4.1(b), first we create a default thread object, *poski_thread*, as a global thread object that is set with *POSKI_THREADPOOL* and 16 threads. Second, we create a tunable-matrix object using the global thread object. Thus *poski_CreateMatCSR* uses *POSKI_THREADPOOL* and 16 threads, and creates 16 sub-matrices if the default partition object is used. Then, we modify only the local thread object in the tunable-matrix object with *POSKI_OPENMP* and 8 threads, while the global thread object *poski_thread* remains the same as the default setting. Here, the given number of threads could be adjusted to satisfy both first and second conditions, although that is not necessary here. In this case, when user calls kernel routines with the tunable-matrix object *A_tunable*, the kernel routines use *POSKI_OPENMP* and 8 threads for 16 sub-matrices in parallel.

The user may call this routine with both arguments *<thread object>* and *<tunable-matrix object>* to modify both global and local thread objects.

4.2.3 Destroying a thread object

The user must call *poski_DestroyThreads()*, shown in Example 3.1, to free the global thread object. This routine also destroys the reusable threads, *threadpool*, if the library creates it. The local thread object is destroyed when the user destroys the tunable-matrix object.

4.3 Partition Handler

This module includes the routines, shown in Table 4.4 and Table 4.5, to create/destroy two partition objects: (1) partition-matrix object of type *poski_partitionarg_t* for partitioning a matrix into sub-matrices, and (2) a partition-vector object of type *poski_partitionVec_t* for partitioning a vector into sub-vectors. Currently, the partition-matrix object contains information about the *<partitioning model>* and *<number of partitions>*, and the partition-vector object contains information about the *<tunable-matrix object>*, *<kernel>*, *<transpose>*, and *<vector property>*.

We define a partition object as global or local. The global partition object can be used multiple times when the user creates multiple tunable-matrix or vector-view objects. The local partition object is a copy of the global partition object in the specified tunable-matrix or vector-view object, and it can be used only for the tunable-matrix or vector-view object.

The actual matrix partitioning occurs at the call to create a tunable-matrix object, and the actual vector partitioning occurs at the call to create a vector-view object or at the call to perform a kernel operation.

Routine	Description
poski_PartitionMatHint	Create a valid, partition-matrix object.
poski_DestroyPartitionMat	Free a partition-matrix object.

poski_report_partitionmodel	Report information of the currently used partitioning model.

Table 4.4: Creating and destroying a partition-matrix object. Bindings appear in Appendix A.3.

Routine	Description
poski_PartitionVecHint	Create a valid, partition-vector object.
poski_DestroyPartitionVec	Free a partition-vector object.

Table 4.5: Creating and destroying a partition-vector object. Bindings appear in Appendix A.4.

4.3.1 Creating a partition object

User optionally call the routine *poski_PartitionMatHint()* or *poski_PartitionVecHint()*, to set a desired partition object instead of using the default partition object. The options for partition-matrix object are shown in Table 4.6, and the options for partition-vector object are shown in Table 4.7. Currently, pOSKI supports only two partitioning models based on the one-dimensional row-wise partitioning scheme. For more detail on the partitioning schemes, see Appendix C.

Hint	Option	Description
<partitioning model>	*OneD	Use one-dimensional row-wise partition by rows.
	SemiOneD	Use one-dimensional row-wise partition by nonzeros.

Table 4.6: Available options for creating the partition-matrix object. The default option is marked by an asterisk (*).

Hint	Option	Description
<vector property>	INPUTVEC	Specify a vector as input of a kernel.
	OUTPUTVEC	Specify a vector as output of a kernel.
<kernel>	See kernel types in Table 4.16.	Specify a kernel
<transpose>	See transpose options in Table 4.17.	Specify a transpose

Table 4.7: Available options for creating the partition-vector object.

To create a desired partition-matrix object, the user must call *poski_PartitionMatHint()* with the user's desired settings. The first argument of this routine is *<partitioning model>* to set the particular partitioning model, and second argument is *<number of partitions>* to set the desired number of partitions to create sub-matrices. To use the desired partition-matrix object, a user must assign it when the user creates a tunable-matrix object as shown in Example 4.2(a). Otherwise, the default partition-matrix object is used to partition a matrix into sub-matrices at the call to create a tunable-matrix object. The default *<partitioning model>* is *OneD*, and the default *<number of partitions>* is the number of threads.

However, the given number of partitions will be automatically adjusted at the call to create a tunable-matrix object. The actual number of partitions should satisfy the following conditions:

(1) The number of partitions should be equal to (*the number of threads* × *k*), where *k* is an integer.

(2) The number of partitions should be less than or equal to the number of rows when the default *<partitioning model>*, *OneD,* is used, or the number of partitions should be less than or equal to the number of non-zeros when the *SemiOneD* is used.

Similarly, the user must call *poski_PartitionVecHint()* to set the desired partition-vector object. A partition-vector object depends on a particular partition-matrix object and kernel operation. The first argument of this routine is *<tunable-matrix object>* to set a particular tunable-matrix object corresponding to the vector, the second and third arguments of this routine are *<kernel>* and *<transpose>* to set a particular kernel operation using the vector, and fourth argument is *<vector property>* to set the usage of the vector for a particular kernel operation. To use a desired partition-vector object, a user must assign it when the user creates a vector-view object as shown in Example 4.2(b). In this case, the routine performs actual vector partitioning into sub-matrices. Otherwise, the default partition-vector object is used to partition a vector into sub-vectors when user calls a kernel routine. If the vector-view-object using the specified partition-vector object is not matching for performing a particular kernel operation at the call the kernel routine, the local partition-vector object of the vector-view object will be automatically modified.
Example 4.2 shows two simple examples of creating a partition-matrix/vector object by calling *poski_PartitionMatHint()* or *poski_PartitionVecHint()*.

Example 4.2: This example illustrates basic partition object creation in pOSKI. Here, we create a partition object with the user's hint for (a) a matrix or (b) a vector.
(a) Create a partition-matrix object

```
poski_partitionarg_t  *partitionMat = poski_PartitionMatHints(SemiOneD, 8, <kernel>, <transpose>);
poski_mat_t  A_tunable = poski CreateMatCSR ( ..., partitionMat, ...);
```

(b) Create a partition-vector object

```
poski_partitionVec_t   *partitionVec =
                    poski_PartitionVecHints(A_tunable, KERNEL_MatMult, OP_NORMAL, INPUTVEC};
poski_vec_t    x_view = poski_CreateVec(..., partitionVec);
```

In Example 4.2(a), we create a partition-matrix object, *partitionMat*, with *<partitioning model>* = *SemiOneD* and *<number of partitions>* = 8. Then, we assign the partition-matrix object at the call to create a tunable-matrix object, *A_tunable*. Here the given partition-matrix object is copied as a local object in *A_tunable*. However, the number of partitions in the local object may be adjusted to satisfy the above conditions. For instance, if the number of threads is 6 in the local thread object and the number of non-zeros is 10, the number of partitions in the local partition-matrix object is set to 6 instead of 8. In this case, total 6 sub-matrices are created.
In Example 4.2(b), we create a partition-vector object, *partitionVec*, with *<tunable-matrix object>* = *A_tunable*, *<kernel>* = *KERNEL_MatMult*, *<transpose>* = *OP_NORMAL*, and *<vector property>* = *INPUTVEC*. Here, *A_tunable*, *KERNEL_MatMult* and *OP_NORMAL* indicate that the vector is used for performing a sparse matrix-vector multiply computation (SpMV) with *A_tunable*. *INPUTVEC* indicates that the vector is used for an input vector of the specified kernel operation. The actual vector partitioning occurs at the call *poski_CreateVec()* to create a vector-view object, *x_view*, using the partition-vector object. Here, the number of partitioned sub-vectors is equal to the number of sub-matrices.

4.3.2 Destroying a partition object

The user must call *poski_DestroyPartitionMat()/poski_DestroyPartitionVec()* to free the partition-matrix/vector object if it was created. The local partition object will be destroyed when the user destroys the tunable-matrix/vector-view object.

4.4 Matrix Handler

This module includes the routines, shown in Table 4.8, to create/modify/destroy a tunable-matrix object of type *poski_mat_t.* Currently, pOSKI supports only an input sparse matrix in CSR format to create a tunable-matrix object.

Routine	Description
poski_CreateMatCSR	Create a valid, tunable-matrix object from arrays of an input sparse matrix in CSR format.
poski_CreateMatCSRFile	Create a valid, tunable-matrix object from a sparse matrix object that the poski_LoadMatrix routine creates from a file.
poski_DestroyMat	Free a tunable-matrix object.
poski_LoadMatrix	Create a valid, sparse matrix object by loading a pattern of non-zeros from a sparse matrix file.
poski_GetMatEntry	Get a value of the specific matrix entry.
poski_SetMatEntry	Set a value of the specific matrix entry.
poski_GetMatSubset	Get a subset of values, specified as a clique or indexed list.
poski_SetMatSubset	Set a subset of non-zero values, specified as a clique or indexed list.

Table 4.8: Creating, modifying and destroying a tunable-matrix object. Bindings appear in Appendix A.5.

4.4.1 Creating a tunable-matrix object

As shown in Example 3.1, the user must call *poski_CreateMatCSR() / poski_CreateMatCSRFile()* to create a tunable-matrix object of type *poski_mat_t* from valid inputs, *<arrays of* a *sparse matrix in CSR format> / <sparse matrix object>*, *<copy mode>*, *<thread object>*, *<partition-matrix object>* and {*matrix properties*}. The argument of *<thread object>* should not be set to *NULL*, but *<partition-matrix object>* might be set to *NULL* for using the default partition-matrix object.

This routine performs the following three major operations: (1) share or copy the input sparse matrix based on *<copy mode>*, (2) copy *<thread object>* and *<partition-matrix object>* as local objects in a tunable-matrix object, and (3) create sub-matrices based on *<partition-matrix object>* in parallel with using *<thread object>*.

To make memory usage logically explicit, the routine supports two data *copy modes*. These modes, defined by the scalar type *poski_copymode_t*, are shown in Table 4.9. The optional {*matrix properties*} for specifying how the library should interpret that data are shown in Table 4.10, and the default properties assumed by the library are marked with an asterisk (*).

Mode	Option	Description
<copy mode>	SHARE_INPUTMAT	User and library agree to share the input sparse matrix arrays.
	COPY_INPUTMAT	The library copies the input sparse matrix arrays, and the user may free them immediately upon return from the handle creation routine.

Table 4.9: Available copy modes for the matrix creation routines.

Property	Option	Description
<non-zero pattern>	*MAT_GENERAL	Input matrix specifies all non-zeros.
<index>	INDEX_ONE_BASED	Array indices start at 1.
	*INDEX_ZERO_BASED	Array indices start at 0 (default C convention).

Table 4.10: Available input matrix properties for the matrix creation routines.

We show two simple examples how to create a tunable-matrix object with a user's input sparse matrix in Example 4.3. The user can set an input matrix using sparse matrix arrays in CSR format or a pattern of non-zeros in a file.

Example 4.3: This example illustrates how to create a basic tunable-matrix object in pOSKI. Here, we create a tunable-matrix object using (a) arrays of a sparse matrix in CSR format or (b) a sparse-matrix object from loading a pattern of non-zeros in a sparse matrix file.

(a) Creating a tunable-matrix object using arrays of a sparse matrix in CSR format

```
int nrows=3;              int ncols=3;              int nnz=5;
int Aptr[4]={0, 1, 3, 5}; int Aind[5]={0, 0, 1, 0, 2}; double Aval[5]={1, -2, 1, 0.5, 1};
poski_mat_t  A_tunable = poski CreateMatCSR ( Aptr, Aind, Aval, nrows, ncols, nnz, <copy_mode>,
                         <thread object>, <partition-matrix object>, <k>, <property_1>, ..., <property_k>);
```

(b) Creating a tunable-matrix object using a sparse-matrix object

```
poski_sparse_matrix_t  *SpA = poski_LoadMatrix(<file name>, <file format>);
poski_mat_t  A_tunable = poski CreateMatCSRFile ( SpA, <copy_mode>, <thread object>,
                         <partition-matrix object>, <k>, <property_1>, ..., <property_k>);
```

In Example 4.3(a), we create the arrays of a sparse matrix in CSR format, then we pass the arrays to create a tunable-matrix by calling *poski_CreateMatCSR()*.

In Example 4.3(b), we call *poski_LoadMatrix()* to create a sparse-matrix object from loading a pattern of non-zeros in a sparse matrix file, then we assign the sparse-matrix object to create a tunable-matrix by calling *poski_CreateMatCSRFile()*. Here, the argument *<k>* indicates the number of assigned properties. The argument *<file name>* indicates the name of a sparse matrix file, and the argument *<file format>* indicates the sparse matrix file format in {*HB, MM*}. Here, *HB* indicates Harwell-Boeing file format, and *MM* indicates Matrix-Market file format. For an input pattern of non-zeros in a file, pOSKI currently supports only Harwell-Boeing sparse matrix file format [3], which is the most popular mechanism for text-file exchange of sparse matrix data.

After creating a tunable-matrix object, the user can modify the local thread object in the <*tunable-matrix object*> while the local partition-matrix object is fixed in <*tunable-matrix object*>. For more detail on how to modify the local thread object, see Section 4.2.

4.4.2 Modifying a tunable-matrix object

The non-zero pattern of the input matrix fixes the non-zero pattern of a tunable-matrix object, *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. To modify an individual value of a matrix, the user must call *poski_SetMatEntry()* with valid inputs:

poski_SetMatEntry (<tunable-matrix object>, <row>, <col>, <value>);

Here, the first argument is the <*tunable-matrix object*> of the user's input matrix *A*, the second and third arguments are the entry position of *row* and *col* in two-dimensional matrix format, and the last argument <*value*> is the new value to be set. If *A_tunable* shares the user's input matrix *A*, the user's values array will be also changed. Logical non-zero values are subject to properties asserted at matrix creation-time. To get the individual value of a matrix, the user must call *poski_GetMatEntry*(<*tunable-matrix object*>, <*row*>, <*col*>), and this call returns the value of *A*(*row,col*). The library also supports changing or getting a subset of values defined by a clique (rectangular subblock) or as a list of non-zero entries with pairs of subscripts. To modify a subset of values, the user may call *poski_SetMatSubset()* or *poski_GetMatSubset()* with valid inputs. For more detail on how to handle a subset of values, see Appendix A.5.
When the tunable-matrix object, *A_tunable*, is tuned, the tuned data structure may store additional explicit zeros to improve performance. The user should avoid changing or getting entries that were not explicitly stored in the input matrix *A* when *A_tunable* was created. The library will report these attempts as errors.

4.4.3 Destroying a tunable-matrix object

The user must call *poski_DestroyMat()*, shown in Example 3.1, to free the tunable-matrix object. This routine also cleans the local thread and partition-matrix objects in the tunable-matrix object.

4.5 Vector Handler

This module includes the routines, shown in Table 4.11, to create/destroy a vector-view object of type *poski_vec_t*. Currently, pOSKI supports a vector-view object only for a single vector.

Routine	Description
poski_CreateVec	Create a valid, vector-view object from an input vector data.
poski_DestroyVec	Free a vector-view object.

Table 4.11: Creating and destroying a vector-view object. Bindings appear in Appendix A.6.

4.5.1 Creating a vector-view object

As shown in Example 3.1, the user must call *poski_CreateVec()* to creates a vector-view object of type *poski_vec_t*, which looks like

poski_vec_t *x_view =*
poski_CreateVec*(<vector data>, <vector length>, <distance>, <partition-vector object>);*

Here, the argument <*vector data*> is an array of the vector data, <*vector length*> is the length of the vector data, and <*distance*> is the distance between logically consecutive elements of the vector data. Currently, pOSKI supports <*distance*> as only *STRIDE_UNIT* as shown in Table 4.12. The last argument <*partition-vector object*> is the partition-vector object.
This routine may perform two or three major operations; (1) share the vector data, (2) copy <*partition-vector object*> as a local object into the vector-view object, and (3) may create sub-vectors based on <*partition-vector object*>. For instance, if <*partition-vector object*> is set to *NULL* as default, no vector partitioning occurs at the call this routine. Otherwise, this routine partitions the vector data into sub-vectors based on the specified <*partition-vector object*>.

Property	Option	Description
<*distance*>	STRIDE_UNIT	Specifying the distance between logically consecutive elements of a vector is 1.

Table 4.12: Available input vector properties for the vector-view creation routine.

4.5.2 Destroying a vector-view object

The user must call *poski_DestroyVec(),* shown in Example 3.1, to free the vector-view object. This routine also cleans the local partition-vector object in the vector-view object.

4.6 Tuning Handler

This module includes the routines, shown in Table 4.13, to tune a tunable-matrix object. The library tunes by selecting a data structure customized for the user's sparse matrix, kernel workload, and machine. The routine defines three groups of tuning operations as the list in Table 4.13.

Routine	Description
poski_TuneHint_Structure	Specify hints about the non-zero structure that may be relevant to tuning. For a list of available hints, see Table 4.14.
poski_TuneHint_MatMult	Workload hints specify the expected options and frequency of the corresponding kernel call.
poski_TuneMat	Tune the matrix data structure using all hints and implicit workload data accumulated so far.

Table 4.13: Tuning primitives for a tunable-matrix object. Bindings appear in Appendix A.7.

4.6.1 Providing explicit structural hints

User may call *poski_TuneHint_Structure()* to provide one or more structural hints, shown in Table 4.14. Providing these hints is entirely optional, but a library implementation may use these hints to constrain a tuning search.

Hint	Option	Arguments	Description
<block>	HINT_NO_BLOCKS	none	Matrix contains little or no dense block substructure.
	HINT_SINGLE_BLOCK	[int r, c]	Matrix structure is dominated by a single block size, $r \times c$.
	HINT_MULTIPLE_BLOCKS	[int k, r_1,c_1, ..., r_k, c_k]	Matrix structure consists of at least $k >= 1$ multiple block sizes, These sizes include $r_1 \times c_1$, ..., $r_k \times c_k$
<align>	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.

Table 4.14: Available structural hints for the matrix tuning routines.

The user should only specify one of the options with appropriate arguments for each hint to provide additional information of the user's sparse matrix structure. For instance, the *<block>* hint, *HINT_SINGLE_BLOCK*, tells the library that the matrix structure is dominated by dense blocks of a particular block size ($r \times c$). The user may (a) set *ARGS_NONE* if the particular block size ($r \times c$) is unknown, or (b) specify the block size ($r \times c$) explicitly if it is known:

(a) ***poski_TuneHint_Structure****(A_tunable, HINT_SINGLE_BLOCKSIZE, ARGS_NONE);*
(b) ***poski_TuneHint_Structure****(A_tunable, HINT_SINGLE_BLOCKSIZE, 6, 6);*

In these cases, either call is "correct" since specifying the block size is optional. However, if more than one option is given for the same hint, the library assumes that the latter option is true. For example, if the user specifies *HINT_SINGLE_BLOCKSIZE* followed by *HINT_NO_BLOCKS*, then no-block option should override the single-block size option for *<block>* hint.

4.6.2 Providing explicit workload hints

The user may call *poski_TuneHint_MatMult()* to tell the library which kernel user will call and with what arguments for a given matrix object, and the expected frequency of such calls. The routine for specifying workload hints all have an argument signature of the form

poski_TuneHint_<kernel> *(A_tunable, {kernel_params}, num_calls);*

where, *num_calls* is an integer. This hint tells the library that the user will call the specified <*kernel*> on the object *A_tunable* with the arguments {*kernel_params*}, and that the user expects to make *num_calls* such calls. Instead of specifying an estimate of the number of calls explicitly, the user may substitute the symbolic constant as shown in Table 4.15. The use of two constants allows a library implementation to provide two level of tuning when the user cannot estimate the number of calls.

Hint	Symbolic constant	Description
<*calls*>	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 4.15: Available symbolic calling frequency constants for <*calls*> hint.

Where a kernel expects a vector-view object to be passed as an argument in {*kernel_params*}, the user may pass to the workload hint one of symbolic constants, shown in Table 4.16, instead of an actual vector-view object. Currently, pOSKI supports only *SYMBOLIC_VEC* for using a single vector.

Hint	Symbolic constant	Description
<*vec*>	SYMBOLIC_VEC	A symbolic single vector-view.

Table 4.16: Available symbolic calling frequency constants for <*vec*> hint.

4.6.3 Explicit tuning

The user must explicitly call the "*tune routine*", *poski_TuneMat()*, to tune a tunable-matrix object. The argument of this routine is a valid <*tunable-matrix object*>. Conceptually, this routine marks the point in program execution at which the library may spend time changing the data structure for each sub-matrix of the tunable-matrix object in parallel. Therefore, each sub-matrix may have a different data structure. This routine also uses any hints about the non-zero structure by calling *poski_TuneHint_Structure()* or workload by calling *poski_TuneHint_MatMult()*.

Example 4.4 shows a simple example of explicit tuning. The first hint, made via a call to *poski_TuneHint_Structure()*, is a *structural hint* telling the library that the user believes that the matrix non-zero structure is dominated by a single block size (*r, c*). For this example, the user might explicitly specify a block size, though here we use 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 of each sub-matrix in parallel. The second hint, made via a call to *poski_TuneHint_MatMult()*, 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, *poski_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. The user could pass an actual instance of a vector-view object that has the precise stride and data layout information.
The actual tuning occurs at the call to *poski_TuneMat*(). This example happens to execute SpMV exactly 500 times, though there is certainly no requirement to do so using symbolic calling frequency constants as shown in Table 4.15.

Example 4.4: This example illustrates basic explicit tuning in pOSKI.

```
poski_mat_t  A_tunable = poski _CreateMatCSR (...);
poski_TuneHint_Structure(A_tunable, HINT_SINGLE_BLOCK, ARGS_NONE);
poski_TuneHint_MatMult(A_tunable, OP_NORMAL, 1, SYMBOLIC_VEC, 1, SYMBOLIC_VEC, 500);
poski_TuneMat(A_tunable);
for (i=0; i<500; i++)
        poski_MatMult(A_tunable, ...);
```

4.7 Kernel Handler

This module includes the routines, shown in Table 4.17, to perform sparse matrix computations. Currently pOSKI supports only a kernel for sparse matrix vector multiply (SpMV) by calling *poski_MatMult()*.
This routine performs three major operations: (1) check partition status of the tunable-matrix and vectors, (2) run an appropriate sparse matrix-vector multiply in parallel, and (3) automatically run the reduction operation in parallel if required. This routine performs the vector partitioning during the first step if the vectors are not properly partitioned for the appropriate kernel with the matrix.

Routine	Description
poski_MatMult	Sparse matrix vector multiply (SpMV) $y = \alpha \bullet op(A)x + \beta \bullet y$, *where op(A) in* $\{A, A^T\}$.

Table 4.17: Available sparse matrix kernels. Bindings appear in Appendix A.8.

We follow the BLAS convention of allowing the user to apply the transpose. Currently supported transpose options provided by the scalar type *poski_transpose_t* are listed in Table 4.18. The notation *op(A)* indicates that any of A or A^T may be applied, where *op(A)* in $\{A, A^T\}$.

Property	Option	Description
<*transpose*>	OP_NORMAL	Apply $op(A) = A$.
	OP_TRANS	Apply $op(A) = A^T$.

Table 4.18: Available transpose options for sparse matrix-vector multiply.

5 Troubleshooting

If user is having difficulty with pOSKI, here are a few things to try to help track down the problem.

5.1 Installation problems

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

5.2 Run-time errors

When running the user's application, if pOSKI reports errors that the user can't otherwise figure out, try setting the environment variable POSKI_MESSAGE_LEVEL to a value of 1 or higher before running the user's application:

% env POSKI_MESSAGE_LEVEL=10 ./user's_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.

5.3 Tuning difficulties

If pOSKI does not seem to be tuning, first try enabling run-time debug messages at level 10. pOSKI's performance tuning heuristics will print data that helps explain why it failed to tune. Keep in mind that pOSKI 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.

References

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

[2] E.-J. Im, K. A. Yelick, and R. Vuduc. *SPARSITY: Framework for optimizing sparse matrix-vector multiply.* International Journal of High Performance Computing Applications, 2004.

[3] I. Duff, R. Grimes, and J. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection.* Technical Report TR/PA/92/86, CERFACS, 1992.

[4] I. Duff, R. Grimes, and J. Lewis. *Sparse Matrix Test Problems.* ACM Transactions on Mathematical Software, 1989.

[5] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. *Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms.* Supercomputing, 2007.

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

[7] R. Vuduc, J. W. Demmel, and K. A. Yelick. *OSKI: A library of automatically tuned sparse matrix kernels.* in Proc. of SciDAC 2005, J. of Physics: Conference Series, San Francisco, CA June 2005

[8] R. Vuduc, J. W. Demmel, and K. A. Yelick. *The Optimized Sparse Kernel Interface (OSKI) Library: User's Guide for Version 1.0.1h.* UC Berkeley, 2007. bebop.cs.berkeley.edu/oski/

[9] R. C. Whaley. *Automatically tuned Linear Algebra Software (ATLAS) home page.* http://math-atlas.sourceforge.net, 2005.

[10] BeBOP: Berkeley Benchmarking and Optimization Group, bebop.cs.berkeley.edu

[11] pOSKI: Parallel Optimized Sparse Kernel Interface. bebop.cs.berkeley.edu/poski

Appendix

A. 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 ≥ 0, returns n! = n x (n-1) x ... x 2 x 1 if n ≥ 1, or 1 if n=0.

Parameters*:*

n [input] *n ≥ 0*
Non-negative integer of which to compute the 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.

A.1 pOSKI library open and close
We summarize the routines for the pOSKI library and present their bindings.

poski_Init	*Initialize pOSKI library.*
poski_Close	*Close poski library.*

int ***poski_Init****();*

Initializes pOSKI library. User must call this function before calling other routines in the library.

Actions and Returns*:*
Returns 0 if the pOSKI library was fully successfully initialized, or an error message on error.

Error conditions and actions:
Possible error conditions include:
1. *If initializing the log file failed [ERR_INIT_LOGFILE]*
2. *If initializing the debug level failed [ERR_INIT_DEBUG]*
3. *If initializing the module loader failed [ERR_INIT_MODULE]*
4. *If initializing the matrix type manager failed [ERR_INIT_MATTYPE]*
5. *If initializing the heuristic manager failed [ERR_INIT_HEURISTIC]*

int ***poski_Close*** *();*

Closes pOSKI library. The library is no longer usable.

Actions and Returns*:*
Returns 0 if the pOSKI library was fully successfully closed, or an error message on error.

Error conditions and actions*:*
Possible error conditions include:
1. *If closing the log file failed [ERR_CLOSE_LOGFILE].*
2. *If closing the module loader failed [ERR_CLOSE_MODULE]*
3. *If closing the matrix type manager failed [ERR_CLOSE_MATTYPE]*
4. *If closing the heuristic manage failed [ERR_CLOSE_HEURISTIC]*

A.2 Threading

We summarize the routines for threading and present their bindings.

poski_InitThreads	*Create a valid thread object.*
poski_ThreadHints	*Modify the thread object.*
poski_DestroyThreads	*Destroy the thread object*

*poski_threadarg_t** ***poski_InitThreads*** *();*

Creates and returns a valid thread object based on #cores on system with the default threading model POSKI_THREADPOOL using POSIX Threads (Pthreads) as a threadpool.
Note: NUMA affinities can be set at this call (future work).

Actions and Returns*:*

A valid <thread object>, or aborts program with reporting an error message on error. Any creating matrix operations, setting thread hints operations, kernel operations, or tuning operations may be called using this object.

Error conditions and actions*:*

Aborts program if initializing thread model failed.
Possible error conditions include:

1. *If invalid threading model is used [ERR_INVALID_THREAD].*
2. *If creating threads as a threadpool failed [ERR_CREATE_THREAD]*

void ***poski_ThreadHints*** *(poski_threadarg_t *poski_thread, poski_mat_t A_tunable, poski_threadtype_t ttype, int nthreads);*

Given thread type in {POSKI_THREADPOOL, POSKI_PTHREAD, POSKI_OPENMP} and number of threads by user, overwrites the parallel context (threading model type and number of threads) in the thread object (poski_thread) or the valid tunable-matrix object (A_tunable).
Note: user's specified OpenMP environment can be set with this call (future work). We can add more threading model such as TBB etc. in thread type (future work)

Parameters*:*

poski_thread [input/output]
A valid <thread object>, or NULL.
A_tunable [input/output]
A valid <tunable-matrix object> of a matrix A, or NULL.
ttype [input]
The <threading model> that the user wants to use in {POSKI_THREADPOOL, POSKI_PTHREAD, POSKI_OPENMP}.
nthreads [input]
The <number of threads> that the user wants to create.

Actions and Returns*:*

Returns 0 if modifying the thread or matrix object was successful, or aborts program with reporting an error message on error.
Adjusts the number of threads to satisfy the following conditions:

1. The number of threads must be between 1 and the number of available cores on the system.
2. If the second argument <*tunable-matrix object*> is given, the number of threads must be equal to

(*the number of partitions / k*), where *k* is an integer.

***Error conditions and actions**:*

Aborts program if overwriting the thread object failed.

Possible error conditions include:

1. *Any of the argument preconditions above are not satisfied [ERR_INVALID_ARG].*
2. *If both thread and matrix objects are NULL [ERR_INVALID_OBJ].*

*int **poski_DestroyThreads** (poski_threadarg_t *thread);*

Frees thread object memory associated with a given thread object. The object is no longer usable. If threadpool was created, this routine also destroys the threadpool.

***Parameters**:*

thread [input]

The valid <thread object>.

***Actions and Returns**:*

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

***Error conditions and actions**:*

Possible error conditions include:

1. *Invalid <thread object> [ERR_INVALID_OBJ].*
2. *If destroying the threads failed [ERR_DESTROY_THREAD]*

A.3 Matrix Partitioning

We summarize the routines for partitioning a matrix and present their bindings.

poski_PartitionMatHints	*Create valid <partition-matrix object> which includes information of partitioning a matrix.*
poski_DestroyPartitionMat	*Destroy the <partition-matrix object>.*

*poski_partitionarg_t** ***poski_PartitionMatHints*** *(poski_partitiontype_t ptype, int npartitions, poski_kernel_t kernel, poski_operation_t op);*

Creates and returns valid matrix partition information based on input parameters.

Parameters*:*

ptype [input]

The partitioning model in {OneD, SemiOneD} which user wants to use.

npartitions [input]

The number of partitions which user wants to create sub-matrices.

kernel, op [input]

The kernel operation which user wants to specify the partition of a matrix for a given kernel operation.

Actions and Returns*:*

A valid <partition-matrix object> or aborts program with reporting an error message on error. Any creating matrix operations may be called using this object.

This routine will automatically adjust the number of partitions to satisfy the following conditions:

(1) The number of partitions should be equal to (the number of threads × k), where k is an integer.

(2) The number of partitions should be less than or equal to the number of non-zeros.

(3) When the default <partitioning model> OneD is used, the number of partitions should be less than or equal to the number of rows.

Error conditions and actions*:*

Possible error conditions include:

1. Any of the argument preconditions above are not satisfied [ERR_INVALID_ARG].

int ***poski_DestroyPartitionMat*** *(poski_partitionarg_t *partitionMat);*

Frees object memory associated with a given <partition-matrix object>. The object is no longer usable.

Parameters*:*

partitionMat [input]

The valid <partition-matrix object>.

Actions and Returns*:*

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

Error conditions and actions*:*

Possible error conditions include:

1. Invalid <partition-matrix object> [ERR_INVALID_OBJ].

A.4 Vector Partitioning

We summarize the routines for partitioning a vector and present their bindings.

poski_PartitionVecHints	*Create a valid <partition-vector object> which includes information of partitioning a vector.*
poski_DestroyPartitionVec	*Destroy the <partition-vector object>.*

*poski_partitionVec_t** ***poski_PartitionVecHints*** *(poski_mat_t A_tunable, poski_kernel_t kernel, poski_operation_t op, poski_vecprop_t vecprop);*

Creates and returns a valid <partition-vector object> which includes information based on input parameters.

Parameters*:*

A_tunable [input]

The valid <tunable-matrix object>of a matrix A.

kernel, op [input]

Specifies the kernel operation for the vector object.

vecprop [input]

Specifies the vector as input or output in {INPUTVEC, OUTPUTVEC}.

Actions and Returns*:*

A valid <partition-vector object> or aborts program with reporting an error message on error. Any creating vector operations or kernel 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_INVALID_ARG].*

int ***poski_DestroyPartitionVec*** *(poski_partitionVec_t *partitionVec);*

Frees object memory associated with a given <partition-vector object>. The object is no longer usable.

Parameters*:*

partitionVec [input]

The valid <partition-vector object>.

Actions and Returns*:*

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

Error conditions and actions*:*

Possible error conditions include:

1. *Invalid <partition-vector object> [ERR_INVALID_OBJ].*

A.5 Matrix object

We summarize the routines for a matrix object and present their bindings.

poski_LoadMatrix	*Load sparse matrix in CSR format from a given file.*
poski_CreateMatCSR	*Create tunable-matrix object from CSR format.*
poski_CreateMatCSRFile	*Create tunable-matrix object from sparse matrix loaded from* ***poski_LoadMatrix.***
poski_DestroyMat	*Free a tunable-matrix object.*
poski_GetMatEntry	*Get the value of a specific matrix entry.*
poski_SetMatEntry	*Set the value of a specific matrix entry.*
poski_GetMatSubset	*Get a subset of values, specified as a clique or indexed list.*
poski_SetMatSubset	*Set a subset of non-zero values, specified as a clique or indexed list.*

poski_sparse_matrix_t ***poski_LoadMatrix*** *(char *filename, poski_filetype_t ftype);*

Given file name and type, creates and returns sparse matrix in CSR format.

Parameters*:*

filename [input] filename!=NULL

The matrix file name.

ftype [input]

The matrix file type in {HB for Harwell-Boeing file format, MM for Matrix-Market file format}.

Actions and Returns*:*

Returns a valid <sparse matrix object> in CSR format or aborts program with reporting an error message on error.

Any creating matrix 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_INVALID_ARG].

2. The given matrix file does not correspond to a valid sparse matrix file [ERR_INVALID_FILE].

poski_mat_t ***poski_CreateMatCSR*** *(poski_index_t *ptr, poski_index_t *ind, poski_value_t *val, poski_int_t nrows, poski_int_t ncols, poski_copymode_t mode, poski_threadarg_t *threads, poski_partitionarg_t *partition, int k, [poski_inmatprop_t property_1, ..., poski_inmatprop_t property_k]);*

Creates and returns a valid tunable-matrix object including partitioned matrices (sub-matrices) from a compressed sparse row (CSR) representation.

Parameters*:*

ptr, ind, val [input]

The input matrix pattern and values must correspond to a valid CSR representation.

nrows, ncols [input]

Dimensions of the input matrix.

mode [input]

Specifies the <copy mode> for the arrays ptr, ind, and val.

threads [input]

The <thread object> which includes the threading information.

partitions [input]

The <partition-matrix object> which includes the partitioning information, or NULL.

k [input]

The number of qualifying properties.

property_1, ..., property_k [input; optional]

The user may assert that the input matrix satisfies zero or more properties listed in Table x on page x. Grouped properties are mutually exclusive, and specifying two or more properties from the same group generates an error. The user must supply exactly k properties.

***Actions and Returns**:*

Returns a valid <tunable-matrix object> or aborts program with reporting an error message on error. Any kernel operations, tuning operations, or setting thread model 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_INVALID_ARG].*
2. *More than 1 property from the same group are specified [ERR_INMATPROP_CONFLICT].*
3. *The input matrix arrays do not correspond to a valid CSR representation [ERR_NOT_CSR].*
4. *The input matrix arrays are incompatible with any of the asserted properties [ERR_INMATPROP_FALSE].*
5. *Invalid <thread object> or <partition-matrix object> [ERR_INVALID_OBJ].*

poski_mat_t ***poski_CreateMatCSRFile*** *(poski_sparse_matrix_t *SpA, poski_copymode_t mode, poski_threadarg_t *threads, poski_partitionarg_t *partition, int k, [poski_inmatprop_t property_1, ..., poski_inmatprop_t property_k]);*

Creates and returns a valid tunable-matrix object including partitioned matrices (sub-matrices) from a compressed sparse row (CSR) representation.

***Parameters**:*

SpA [input]

The input <sparse matrix object> in CSR format.

mode [input]

Specifies the <copy mode> for the arrays ptr, ind, and val.

threads [input]

The <thread object> which includes the threading information.

partitions [input]

The <partition-matrix object> which incudes the partitioning information, or NULL.

k [input]

The number of qualifying properties.

property_1, ..., property_k [input; optional]

The user may assert that the input matrix satisfies zero or more properties listed in Table x on page x. Grouped properties are mutually exclusive, and specifying two or more properties from the same group generates an error. The user must supply exactly k properties.

***Actions and Returns**:*

Returns a valid <tunable-matrix object> or aborts program with reporting an error message on error. Any kernel operations, tuning operations, or setting thread model 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_INVALID_ARG].*
2. *More than 1 property from the same group are specified [ERR_INMATPROP_CONFLICT].*
3. *The input <sparse matrix object> does not correspond to a valid CSR representation [ERR_INVALID_CSR].*
4. *The input <sparse matrix object> is incompatible with any of the asserted properties [ERR_INMATPROP_FALSE].*
5. *Invalid <thread object> or <partition-matrix object> [ERR_INVALID_OBJ].*

int ***poski_DestroyMat*** *(poski_mat_t A_tunable);*

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

Parameters*:*

A_tunable [input]

The <tunable-matrix object> of a matrix A.

Actions and Returns*:*

Returns 0 if the <tunable-matrix object> memory was fully successfully freed, or an error code on error.

Error conditions and actions*:*

Possible error conditions include:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*

poski_vaule_t ***poski_GetMatEntry****(poski_mat_t A_tunable, poski_int_t row, poski_int_t col);*

Returns the value of the specified matrix element in A(row, col).

Parameters*:*

A_tunable [input]

The <tunable-matrix object> of a matrix A.

row, col [input]

Specifies the element whose value is to be returned.

Actions and Returns*:*

Returns the value of A(row, col), or an error code on error.

Error conditions and actions*:*

Possible error conditions include:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*
2. *Invalid element (row, col) [ERR_INVAILD_ELEMENT].*

int ***poski_SetMatEntry****(poski_mat_t A_tunable, poski_int_t row, poski_int_t col, poski_value_t val);*

Changes the value of the specified matrix element in A(row, col).

Parameters*:*

A_tunable [input/output]

The <tunable-matrix object> of a matrix A.

row, col [input]

Specifies the element whose value is to be modified.

val [input]

The specified value to modify.

***Actions and Returns**:*

Returns 0 if setting A(row, col) = val is successful, or an error code on error.

***Error conditions and actions**:*

Possible error conditions include:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*
2. *Invalid element (row, col) [ERR_INVAILD_ELEMENT].*

*int **poski_GetMatSubset**(poski_mat_t A_tunable, int numRows, int numCols, int rowStride, int colStride, poski_int_t *rows, poski_int_t *cols, poski_value_t *vals, poski_subsettype_t subsettype);*

Returns the values, defined by a subset, in a matrix.

***Parameters**:*

A_tunable [input]

The <tunable-matrix object> of a matrix A.

numRows, numCols [input]

Number of rows or columns in the subset.

rowStride, colStride [input]

Stride between rows or columns of vals.

rows, cols [input]

Row or Column indices of the subset.

vals [input/output]

The values, defined by a subset, stored as an array.

subsettype [input]

The type of subset in {POKSI_BLOCKNENTRIES, POSKI_ARRAYENTRIES}.

***Actions and Returns**:*

Returns values in the subset.

If <subsettype> is set as POSKI_BLOCKENTRIES, this routine returns X(r,c) = A(i,j), where i=rows[r] and j=cols[c], for all 0≤ r <numRows and 0≤ c <numCols. Here, X is the numRows×numCols matrix corresponding to vals.

If <subsettype> is set as POSKI_ARRAYENTRIES, this routine makes an attempt to return vals[r] = A(i,j), where i=rows[r] and j=cols[r], for all 0 ≤ r <numRows.

This routine will set X(r,c) or vals[r] to zero, and report an error message for each invalid entry A(i,j) which is not explicitly stored when A_tunable was created. This routine also reports an error message for each invalid entry A(i,j) which is out-of-range.

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

***Error conditions and actions**:*

Possible error conditions include:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*
2. *Invalid <subsettype> [ERR_INVAILD_TYPE].*

int ***poski_SetMatSubset****(poski_mat_t A_tunable, int numRows, int numCols, int rowStride, int colStride, poski_int_t *rows, poski_int_t *cols, poski_value_t *vals, poski_subsettype_t subsettype);*

Changes the values, defined by a subset, in a matrix.

Parameters*:*

A_tunable [input/output]

The <tunable-matrix object> of a matrix A.

numRows, numCols [input]

Number of rows or columns in the subset.

rowStride, colStride [input]

Stride between rows or columns of vals.

rows, cols [input]

Row or Column indices of the subset.

vals [input]

The values, defined by a subset, stored as an array.

subsettype [input]

The type of subset in {POKSI_BLOCKENTRIES, POSKI_ARRAYENTRIES}.

Actions and Returns*:*

Sets any entry in the subset, which was explicitly stored when A_tunable was created.

If <subsettype> is set as POSKI_BLOCKENTRIES, this routine makes an attempt to set A(i,j) = X(r,c), where i=rows[r] and j=cols[c], for all 0≤ r <numRows and 0≤ c <numCols. Here, X is the numRows×numCols matrix corresponding to vals.

If <subsettype> is set as POSKI_ARRAYENTRIES, this routine makes an attempt to set A(i,j) = vals[r], where i=rows[r] and j=cols[r], for all 0 ≤ r <numRows.

This routine will report an error message for each invalid entry A(i,j) which is not explicitly stored when A_tunable was created or which is out-of-range.

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 <tunable-matrix object> [ERR_INVALID_OBJ].*
2. *Invalid subsettype [ERR_INVAILD_TYPE].*

A.6 Vector object

We summarize the routines for a vector object and present their bindings.

poski_CreateVec	*Create a valid vector object.*
poski_DestroyVec	*Free a vector object.*

poski_vec_t ***poski_CreateVec*** *(poski_value_t *x, poski_int_t length, poski_int_t inc, poski_partitionVec_t *partitionvec);*

Creates and returns a valid vector-view object on a single dense column vector x. If partitionvec==NULL, no partitioning occurs.

Parameters*:*

x [input]

The input vector x (A pointer to the user's dense array representation of the vector x).

length [input]

Number of vector elements.

inc [input]

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.

partitionvec [input]

The <partition-vector object> which includes information of the vector partition, or NULL for default.

Actions and Returns*:*

A valid <vector-view object> or aborts program with reporting an error message on error. Any kernel operation 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_INVALID_ARG].*

int ***poski_DestroyVec*** *(poski_vec_t x_view);*

Frees object memory associated with a given <vector-view object>. The object is no longer usable.

Parameters*:*

x_view [input]

The <vector-view object> of a vector.

Actions and Returns*:*

Returns 0 if the <vector-view object> memory was fully successfully freed, or an error code on error.

Error conditions and actions*:*

Possible error conditions include:

1. *Invalid <vector-view object> [ERR_INVALID_OBJ].*

A.7 Matrix Tuning

We summarize the routines for tuning a matrix and present their bindings.

poski_TuneHint_Structure	*Specify hints about the non-zero structure.*
poski_TuneHint_MatMult	*Workload hints specify the expected options.*
poski_TuneMat	*Tune the matrix data structure.*

int ***poski_TuneHint_Structure*** *(poski_mat_t A_tunable, poski_tunehint_t hint [, ...]);*

Registers a hint about the matrix structure with a <tunable-matrix object>.

Parameters*:*

A_tunable [input/output]

The <tunable-matrix object> of a matrix A for which to register a structural hint.

hint [input]

User-specified structural hint. This hint may be followed by optional arguments.

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:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*
2. *Specifying a hint with the wrong hint arguments [ERR_BAD_HINT_ARG].*

int ***poski_TuneHint_MatMult*** *(poski_mat_t A_tunable, poski_operation_t op, poski_value_t alpha, poski_vec_t x_view, poski_value_t beta, poski_vec_t y_view, int num_calls);*

Workload hint for the kernel operation poski_MatMult which computes $y = \alpha \bullet op(A)x + \beta \bullet y$*, where op(A) in* $\{A, A^T\}$.

Parameters*:*

A_tunable [input/output]

The <tunable-matrix object> of a matrix A.

op [input]

Specifies op(A).

alpha, beta [input]

Scalar constants α and β, respectively.

x_view, y_view [input]

The valid <vector-view object> for a vector x and y, respectively.

num_calls

The number of times this kernel operation will be called with these arguments.

Actions and Returns*:*

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

Error conditions and actions*:*

Possible error conditions include:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*

2. *Invalid <vector-view object> [ERR_INVALID_OBJ].*
3. *Incompatible operand dimensions [ERR_DIM_MISMATCH].*

int ***poski_TuneMat*** *(poski_mat_t A_tunable);*

Tunes the <tunable-matrix object> using all hints and implicit profiling data.

Parameters*:*

A_tunable [input/output]

The <tunable-matrix object> of a matrix A to tune.

Actions and Returns*:*

Returns a non-negative status code whose possible values are defined by the constants lists, or an error code otherwise.

Error conditions and actions*:*

Possible error conditions include:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*

A.8 Kernel

We summarize the available kernels, and present their bindings.

poski_MatMult	*Sparse matrix-vector multiply (SpMV)* *$y = \alpha \bullet op(A)x + \beta \bullet y$, where op(A) in $\{A, A^T\}$.*

int ***poski_MatMult*** *(poski_mat_t A_tunable, poski_operation_t op, poski_value_t alpha, poski_vec_t x_view, poski_value_t beta, poski_vec_t y_view);*

Computes $y = \alpha \bullet op(A)x + \beta \bullet y$, where op(A) in $\{A, A^T\}$. If a vector is not partitioned or not properly partitioned, the vector partitioning occurs before computing kernel at this call. If a reduction operation for sub-matrices is required, it will occur at end of this call.

Parameters*:*

A_tunable [input]

The <tunable-matrix object> of a matrix A.

op [input]

Specifies op(A).

alpha, beta [input]

Scalar constants α and β, respectively.

x_view, y_view [input]

The valid <vector-view object> for a vector x and y, respectively.

Actions and Returns*:*

Computes $y = \alpha \bullet op(A)x + \beta \bullet y$, where op(A) in $\{A, A^T\}$ and return 0 only if the dimensions of 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:

1. *Invalid <tunable-matrix object> [ERR_INVALID_OBJ].*
2. *Invalid <vector-view object> [ERR_INVALID_OBJ].*
3. *Incompatible operand dimensions [ERR_DIM_MISMATCH].*

B. Threading Models

Currently, pOSKI supports three simple threading models, shown in Example B.1:
(1) POSKI_THREADPOOL using POSIX Threads (Pthreads) as a threadpool, (2) POSKI_PTHREAD using POSIX Threads (Pthreads), and (3) POSKI_OPENMP using OpenMP. The default threading model is POSKI_THREADPOOL, and the default number of threads is the number of available cores on the system. The POSKI_THREADPOOL tells the library to create reusable threads once and may use them multiple times to perform multiple tasks, while other threading models tell the library to create and destroy threads at each task.

Example B.1: This example illustrates basic pseudo codes of threading models to compute SpMV in pOSKI.

(a) POSKI_THREADPOOL: poski_MatMult_threadpool (A, op, alpha, x, beta, y)

```
set nthreads to A->threadargs.nthreads
set threadpool to A->threadargs.thread
set kernel to A->kernel
for i=0 to nthreads-1
        set threadpool[i].Job to KERNEL_MatMult
        set threadpool[i].kernel to kernel[i]
end for
wait for setting the threadpool arguments for all threads
for i=0 to nthreads-1 do in parallel using threadpool
        compute SpMV(kernel[i]) per each thread
end for
Wait for finishing tasks for all threads
```

(b) POSKI_PTHREAD: poski_MatMult_pthread (A, op, alpha, x, beta, y)

```
set nthreads to A->threadargs.nthreads
set kernel to A->kernel
for i=0 to nthreads-1 do in parallel using Pthreads
        compute SpMV(kernel[i]) per each thread
end for
Wait for finishing tasks for all threads
```

(c) POSKI_OPENMP: poski_MatMult_openmp (A, op, alpha, x, beta, y)

```
set nthreads to A->threadargs.nthreads
set npartitions to A->partitionargs.npartitions
set kernel to A->kernel
for i=0 to npartitions-1 do in parallel using OpenMP with num_threads(nthreads)
        compute SpMV(kernel[i]) per each partition
end for
```

pOSKI supports three threading models, as shown in Example B.1, for the following purposes: (1) how to handle thread creation and destruction, and (2) how to schedule sub-matrices to threads and map threads to cores.

B.1 How to handle thread creation and destruction in pOSKI?

When the user repeatedly calls pOSKI routines, the POSKI_THREADPOOL threading model may have slightly better performance than others by avoiding the overhead of creating/destroying threads at each task. However, the threads in the threadpool, which are using thread affinity and a spin lock, may conflict with user's own threads to share hardware resources.
When the user uses threads for other purposes between pOSKI calls, the user may use the POSKI_PTHREAD or POSKI_OPENMP threading model to avoid the overhead of using thread affinity and spin locks in the POSKI_THREADPOOL threading model.

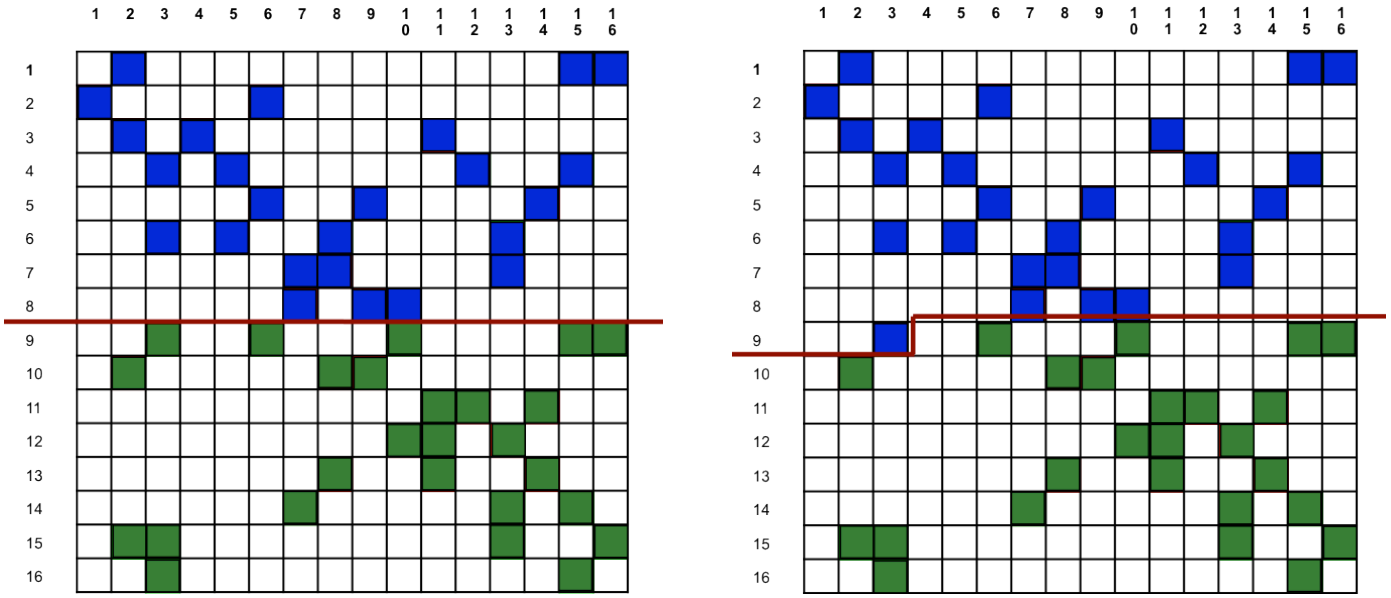
B.2 How to handle schedule submatrices to threads and map threads to cores in pOSKI?
pOSKI currently handles scheduling and mapping based on basic NUMA-aware optimization as follows:

(a) If the user's system supports the *libnuma* library, then pOSKI will try to use *libnuma* routines as default for NUMA-aware optimization.
    - Such as *numa_set_membind()*, *numa_run_on_node()*
(b) If libnuma is not supported, then pOSKI uses a first-touch policy for thread affinity.
    - Such as *CPU_SET()*, *pthread_setaffinity_np()*, *sched_setaffinity()*
(c) The sub-matrices will be scheduled into threads based on cyclic distribution.
    - For example, sub-matrix *i* will be assigned to thread *tid=i%nthreads*, where $0<i\leq npartitions$, *npartitions* is the number of sub-matrices, and *nthreads* is the number of threads.

In the POSKI_THREADPOOL and POSKI_PTHREAD threading models, each sub-matrix uses static scheduling and mapping to a core based on NUMA-aware optimization. However, in the POSKI_OPENMP threading model, each sub-matrix uses default static scheduling in round-robin fashion based on OpenMP's policy, which may violate the pOSKI's NUMA-aware optimization.

C. Partitioning Models

Currently, pOSKI supports two simple partitioning schemes, shown in Figure C.1, based on one-dimensional partition scheme: (a) *One-dimensional row-wise partition by rows* (*OneD*), and (b) *One-dimensional row-wise partition by nonzeros* (*SemiOneD*). We describe the load-balancing approach of both partitioning schemes, which try to assign an equal number of nonzero entries to each partition.



(a) OneD: *One-Dimensional partition by rows* (b) SemiOneD: *One-Dimensional partition by nonzeros*

Figure C.1: Partitioning techniques supported in current pOSKI library.

In Figure C.1, we partition the 16×16 matrix into two sub-matrices. Here, the blue box represents the non-zeros in first sub-matrix, and the green box represents the non-zeros in the second sub-matrix. In this particular examples, both schemes show good load balance; the total of 51 non-zeros of the matrix are divided into 25 and 26 non-zeros for two sub-matrices.

The *OneD* partitioning scheme is very well suited to dense matrices and matrices with uniform sparsity pattern. However, it may be hard to achieve good load balance on sparse matrices with a non-uniform sparsity pattern because of the restriction of rectilinear splits on rows as shown in Figure C.1(a). This scheme also limits the number of partitions by the number of rows.

The *SemiOneD* partitioning scheme is used to solve the load-balancing problem more accurately than the *OneD* partitioning scheme. However, *SemiOneD* may require additional reductions since it may split a single row into several sub-matrices as shown in Figure C.1(b). The maximum number of partitions in this scheme is equal to the number of non-zeros in the matrix.

A summary of the partitioning schemes is shown in Table C.1. We plan to support other partitioning schemes, like two-dimensional jagged-like partition, and graph or hyper-graph partitions, in a later version of pOSKI. The two-dimensional partitioning schemes may require reduction operations in either Ax or A^Tx with same data structure to consider NUMA-aware optimization for both. We also plan to support these partitioning

schemes with (1) reordering and (2) mapping with NUMA-aware optimization that may not require reduction operations for both Ax and A^Tx.

		OneD	*SemiOneD*
Load balance		*nnz / npartitions*	*nnz / npartitions*
Condition		*npartitions <= nrows, and npartitions = k×nthreads, where k=1,2..*	*npartitions <= nnz, and npartitions = k×nthreads, where k=1,2..*
Search space		*O(nrows)*	*O(nrows)*
Required extra space		*Ω(nrows)*	*Ω(nrows)*
Reordering		*No*	*No*
Required reduction	***op(A) = A***	*No*	*Yes*
	op(A) = A^T	*Yes*	*Yes*

Table C.1: The summary of two simple partitioning schemes supported in current pOSKI library.