SparseX

User's Guide

by
Athena Elafrou

June, 2014

Contents

1

Getting started

1.1 About SparseX

The SparseX library is a collection of low-level primitives written in the C/C++ programming languages, that provides the means to developers of solver libraries and of scientific and engineering applications to easily attain high performance of the Sparse Matrix-by-Vector multiplication kernel (SpMV) on modern multicore architectures.

1.2 Goals and motivation

Most sparse iterative solver libraries for linear systems only support standard sparse matrix storage formats, such as the CSR and COO formats, thus exhibiting poor performance in many cases. However, some recent projects that focus on high performance computing (HPC), including the open-source Portable Extensible Toolkit for Scientific computation (PETSc) [Balay et al., 2013] and Intel's commercial Math Kernel Library (MKL) [Intel® Coorporation, 2013], have expanded their suite of sparse matrix representations with more elaborate formats, such as the BCSR format, which are much more efficient for particular types of problems.

The *Compressed Sparse eXtended (CSX)* format, along with its symmetric variant, are currently among the most highly-optimized sparse matrix storage formats for performing matrix-by-vector multiplications on multicore architectures, especially in the context of iterative methods for the solution of large-scale linear systems [Kourtis et al., 2011; Karakasis et al., 2013]. Its unique performance stability across a wide variety of problems makes CSX an excellent candidate for HPC applications and, thus, we believe CSX's integration in the numerical software stack would have an important impact on several scientific applications whose overall performance is sensitive to that of the SpMV kernel. To this end, we provide the means to facilitate this process through a low-level interface. Key goals and aspects of our interface sum up to the following:

Provide simple and clear semantics. Every well-designed API should be easy to use correctly and difficult to use incorrectly. A user-friendly syntax reduces the time

and intellectual overhead required to develop user software as well as making the development process less error prone. Of course, this is a universal objective in interface design and achieving it depends, to a significant degree, in minimizing the number of things the user must remember in order to effectively use the interface. In the present context, this implies:

- the number of function names the user must remember should be small.
- to the extent possible, the information that functions require as input parameters should be limited to information that the user would necessarily know.

Our interface tries to reflect the above "guidelines" as well as being as *intuitive* as possible; that is, usage of the interface should follow the user's natural train of thought on solving the problem. This objective is usually complicated by the desire to serve users with different levels of expertise.

Facilitate integration to large scale sparse solver libraries. Even though the library can be used directly in applications that involve sparse matrix-by-vector multiplications, its ultimate goal is to integrate readily into application-level libraries that provide high-level sparse kernel support (including iterative solution methods of sparse linear systems), such as the aforementioned PETSc library. Integration in such systems has a number of advantages, including the ability to hide data structure details from the user and, of course, the large potential user base that will assist in the better dissemination of the CSX format.

Transparently adjust to the target platform. The SparseX library currently supports symmetric shared memory (SMP) and non-uniform memory access (NUMA) multicore architectures. Any architecture specific optimization is performed transparently during the installation process of the library, eliminating any need for the user to provide information on the hardware platform.

Allow for user inspection and control of the tuning process. Tuning refers to the preprocessing phase of the CSX format, i.e, converting the original sparse matrix into the CSX format. A number of parameters can be explicitly set by the user in order to control different aspects of this phase, such as the number of threads used, defining specific substructure types to search for in the matrix, selecting between CSX or CSX-sym (its symmetric variant) as the target format and so on. This adds a lot of flexibility to the tuning process and also affects performance in a direct manner. For example, if the user is aware of the sparsity structure of the matrix (e.g, consisting mainly of blocks) she can guide the detection process by selecting the block substructure types, reducing the execution time of this phase. Of course, a "poor" selection may result in a significant performance degradation, thus the user is advised to rely on the autotuning capabilities of CSX and

only override an option when prior knowledge is available, as in the example described previously.

For a detailed discussion on the basic elements of the library design and some general implementation details refer to [Elafrou, 2013].

2

Installation

2.1 Installation requirements

In order to install and run SparseX, your system must meet the following requirements:

- A fairly recent Linux OS
- gcc/g++-4.6, gcc/g++-4.7, gcc/g++-4.8
- LLVM == 3.0 and Clang
- Boost Library >= 1.48 (regex, serialization, system, thread)
- numactl library >= 2.0.7

Currently, SparseX has been tested on the following environments:

- **Architectures**: Intel Xeon X7460, Intel Xeon X5560, Intel Xeon X5650, Intel Xeon E5-4620
- **C/C++ Compilers**: gcc/g++-4.6.2, gcc/g++-4.7.2, gcc/g++-4.8.2
- **OS**: Linux kernel 3.7.10

2.2 How to install SparseX

Step 1: Download and extract

In order to use SparseX you must either explicitly download and extract its source code from the associated git link or clone the git repository.

For the first option, a tarball of the latest release can be downloaded from http://research.cslab.ece.ntua.gr/sparsex and subsequently extracted by typing:

$ **tar -xzf** sparsex-x.y.z.tar.gz

These commands unpack the SparseX distribution into a subdirectory named `sparsex-x.y.z`. Replace the "x.y.z" string with the latest available version of SparseX (*major.minor.patch* software versioning scheme).

For the second option, '*cd*' to a directory of your choosing and simply type:

$ **git clone** git://github.com/cslab-ntua/sparsex.git

Note that there is no need to create a new directory `sparsex`, as cloning the repository via git will do that for you.

Step 2: Compile

The simplest way to compile this package includes the following steps

1. If the library has been downloaded in means of cloning the git repository or downloading the tarball available on the github link, you must first run:

   $ **autoreconf -vi**

   in order to remake all of the configure scripts.

2. '*cd*' to the directory containing the package's source code, e.g. sparsex-x.y.z, and type '*./configure*' to configure the package for your system. If you have installed LLVM/Clang in a non-standard location that is not in your path, you can instruct the compilation process to use your preferred LLVM installation through the '*--with-llvm*' configuration option (by providing the absolute path to the LLVM configuration script "llvm-config"). Similarly for the Boost library you can use the '*--with-boostdir*' option.

   $ **cd** SPARSEX_DIR

   $ **./configure** [options]

3. Type '*make*' to compile the package. You can speed up the compilation process by using multiple tasks with the '*-j*' flag of make:

   $ **make -j**8

4. Type '*make install*' to install the library and any data files and documentation. When installing into a prefix owned by root, it is recommended that the package be configured and built as a regular user, and only the 'make install' phase executed with root privileges. By default, 'make install' installs the library under `/usr/local/lib` and include files under `/usr/local/include`. You can specify an installation prefix other than `/usr/local` by giving 'configure' in step 1 the option '*--prefix*=PREFIX', where PREFIX must be an absolute directory path.

   $ **make install**

Customizing the SparseX 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:

$ **./configure --help**

and also refer to the INSTALL document available in the SparseX directory.

Selecting a different build directory. The SparseX library may be built in a different directory simply by creating the directory and running the commands in Step 2 of the previous section from that directory.

Selecting different data type precisions. The SparseX library is defined in terms of two basic types: `spx_index_t` for indexing data and `spx_value_t` for floating point values. These are bound by default to int and double, respectively, but may be overidden at library build-time through the '*--index-type*=<TYPE>' and '*--value-type*=<TYPE>' options of '*configure*'.

Benchmarking. The SparseX distribution also includes a benchmarking framework in order to evaluate the SpMV performance. This framework also supports other popular libraries that implement the SpMV kernel, including Intel®'s Math Kernel Library (Intel® MKL) [Intel® Coorporation, 2013], a well established commercial product, and the parallel Optimized Sparse Kernel Interface (pOSKI) library [Vuduc et al., 2005; Ankit, 2008], which has been developed by the Berkeley Benchmarking and Optimization (BeBOP) group (http://bebop.cs.berkeley.edu/). You can enable this feature through the '*--enable-bench*' option of '*configure*'. If one of the above libraries is installed on your system you can use the '*--with-mkl*=<DIR>' or '*--with-poski*=<DIR>' options respectively to enable them.

After you have successfully compiled the benchmarking framework, the final executable, named '*bench_spmv*', will be placed inside the `src/bench` directory of the build directory. The '*bench_spmv*' executable may be invoked as follows:

$ [**ENV**=<value>] **./bench_spmv -f** <matrix_file> [**-l** <library_name>]

where <library_name> can be one of {SparseX, MKL, pOSKI}.

Execution is controlled by a set of environment variables, namely OUTER_LOOPS, LOOPS and NUM_THREADS. OUTER_LOOPS defines the number of repetitions, LOOPS the number of SpMV iterations and NUM_THREADS the number of threads that will be used. For the SparseX and MKL libraries you can also use CPU_AFFINITY and GOMP_CPU_AFFINITY respectively to set the thread affinity. The pOSKI library does not support explicit configuration of thread affinity.

2.3 Linking against SparseX

If you ever happen to want to link against the library, you must either use libtool, and specify the full pathname of the library, or use the '**-L**LIBDIR' flag during linking and do at least one of the following:

- add LIBDIR to the 'LD_LIBRARY_PATH' environment variable during execution
- add LIBDIR to the 'LD_RUN_PATH' environment variable during linking
- use the '-Wl,-rpath -Wl,LIBDIR' linker flag

- have your system administrator add LIBDIR to ‘/etc/ld.so.conf’

3

Interface

3.1 Overview

The API primitives of the SparseX library fall into two broad categories: the *auxiliary routines* and the *computational routines*. The auxiliary routine set includes

- sparse matrix and vector creation and update;
- sparse matrix tuning into the CSX format;
- sparse matrix and vector reordering;
- CSX update;
- CSX I/O.

The computational routine set follows the "look-and-feel" of the BLAS interface, including

- sparse matrix-by-vector multiplication;
- vector operations (scale, add, subtract, multiply).

The following sections proceed with a thorough presentation of the available routines, both auxiliary and computational, and describe essential ingredients of the user interface, including the major *objects* that are intergal parts of it. Some source code snippets are provided to give a more practical view of how the API may be used. More elaborate examples are provided in th following chapter. For complete C bindings of the available routines see Appendix A.

3.1.1 Naming convention

The naming convention for the public interface routines of the SparseX library has the following form:

(return value type) spx_**object**_*function*(...)

Every routine starts with the **spx** prefix, which stands for **Sp**arse**X**, followed by the name of the object it is associated with (usually in a condensed form) and a name that describes its functionality.

3.1.2 Basic scalar types and objects

The SparseX library is defined in terms of two basic types: `spx_index_t` for indexing data and `spx_value_t` for floating point values. These are bound by default to `int` and `double`, respectively, but may be overidden at library build-time as described in Chapter 2.

Matrices and vectors are represented by handles in our interface. The use of handles complements the object-oriented approach of the core library and enables information hiding. Once created, a matrix or vector is referenced only by its handle. The available handles for manipulating matrices and vectors are defined by the following data types:

- `spx_input_t`, which represents a handle to the input matrix;
- `spx_matrix_t`, which represents a handle to the matrix in the CSX format;
- `spx_vector_t`, which represents a handle to a dense vector object.

We must note here that since our API aims at exporting utilities for the CSX format alone, matrices stored in different formats are only treated as "input" matrix representations, hence the distinction between the `spx_input_t` and `spx_matrix_t` handles.

Complementary handle types include the `spx_partition_t` type, which describes a partitioning object that is required for multithreaded execution, the `spx_perm_t`, for defining permutations in case the user wants to apply reordering to improve the sparsity structure of the matrix and the `spx_timer_t`, which descibes a timer object. These types will be described in more detail in the following sections.

3.2 Auxiliary routines

3.2.1 Loading input matrices

The user creates an input matrix object of type `spx_input_t` from a valid sparse matrix in one the following formats:

Matrix Market File (MMF) format. In dealing with issues of I/O, SparseX is currently designed to support reading a sparse matrix from a file in the MMF format [Boisvert et al., 1996]. File input is embedded as another form of a sparse matrix constructor. A MMF file consists of four parts:

1. **Header line**: this is the first line in the file and contains an identifier and four text fields in the following form

   %%MatrixMarket *object format field symmetry*

   where *object* is either **matrix** (this is the case we will consider here) or **vector**, *format* can be either **coordinate** for sparse matrices or **array** for dense matrices, *field* is either **real**, **double**, **complex**, **integer** or **pattern** and, finally, *symmetry* is either **general**, **symmetric**, **skew-symmetric** or **hermitian**.

Routine	Description
`spx_input_load_mmf`	Creates an input object from a file on disk in the MMF format.
`spx_input_load_csr`	Creates an input object from a matrix in the CSR format.
`spx_input_destroy`	Destroys an input object.

Table 3.1: Available routines for creating and destroying input matrix objects of type `spx_input_t`.

2. **Comment lines**: begin with a percent sign and allow a user to store information and comments.
3. **Size line**: specifies the number of rows, columns and nonzero elements in the matrix.
4. **Data lines**: specify the location of the matrix entries and their values. When the matrix is sparse, the location of the matrix entries is given in the *coordinate* format using one-base indexing in a column-wise ordering.

Since CSX operates on the elements of a sparse matrix in a row-wise order, the column-wise ordering of the MMF format creates the need to sort the elements when loading the matrix. Thus, the format has been extended in order to also support row-wise ordering of the elements and zero-base indexing by introducing two optional fields in the header line called *indexing*, which can be either **0-base** or **1-base**, and *ordering*, which can be either **column** or **row**. Furthermore, a simplified version of the MMF format is supported, which drops the header line and includes only the size line and data lines. In this case, however, the data must be sorted both row- and column-wise.

Compressed Sparse Row (CSR) format. An input handle can also be created from an existing user-allocated, pre-assembled matrix in the CSR format. The CSR data structures (rowptr, colind and values) are "shared" in this case with the library, thus the user must guarantee they will not be freed or reallocated before the destruction of the input handle. Both zero-based and one-based indexing is supported by setting the appropriate argument in the corresponding routine.

Table 3.1 summarizes the available routines for loading input matrices.

3.2.2 Tuning to the CSX format

The user may convert the input matrix into the CSX format simply by calling the `spx_mat_tune()` routine. Even though the parameters of the preprocessing phase

have been expertly tuned, the user can also experiment with the capabilities of CSX through the `spx_set_option()` routine. There are options for controlling: (a) the runtime environment, (b) the preprocessing phase of CSX and (c) the CSX format itself. The available options with their default values for every category are given in Table 3.2.

The options for setting the runtime environment include the number of participating threads (`spx.rt.nr_threads`) and the processor affinity (`spx.rt.cpu_affinity`). If these options are not explicitly set by the user the library automatically detects the optimal configuration, i.e., the number of threads is set to the number of available cores and the CPU affinity is adjusted according to a 'share-nothing' core-filling policy, which assigns threads to cores so that the least resource sharing is achieved.

The user can also control different aspects of the preprocessing phase. For one, she may select specific substructure types to be encoded through the `spx.preproc.xform` option. The {`h,v,d,ad,br,bc`} values correspond to horizontal, vertical, diagonal, anti-diagonal, row-aligned block and column-aligned block substructures respectively. In the case of "`none`", only delta units will be encoded. She can also select the heuristic that will determine the scale of compression: "`ratio`" will focus solely on maximizing the compression ratio, while "`cost`" will also try to minimize the computational cost of the kernel. The first heuristic is the default option in SMP architectures where lower memory bandwidths require a minimal memory footprint, while the second heuristic is the default option for NUMA architectures where increased memory bandwidths might expose computationally intensive loads. Furthermore, the user can enable the use of statistical sampling in the detection process (`spx.preproc.sampling`). The preprocessing cost of CSX can be significantly halved by enabling the use of sampling. There are two sampling methods available: "`portion`" and "`window`". In the first case, the user can provide the portion of the matrix that she wishes to be sampled (`spx.preproc.sampling.portion`) and the number of sampling windows used per thread (`spx.preproc.sampling.nr_samples`) and the autotuning capabilities of CSX will automatically adjust the window size according to the following equation:

$$\begin{aligned} window\_size &= \frac{sampling\_portion \cdot nr\_nzeros_{per\_thread}}{nr\_samples} \\ &= \frac{sampling\_portion \cdot nr\_nzeros_{total}}{nr\_samples \cdot nr\_threads} \end{aligned} \tag{3.1}$$

In the second case, the user will have to explicitly set the window size to a meaningful number of nonzero elements (`spx.preproc.sampling.window_size`). In both cases, if the number of samples is not explicitly set, the default number will be used.

Equation 3.1 designates that the window size has an inverse relation to the number of threads for fixed values of the sampling portion and the number of samples. As the number of threads increases it is possible that the sampling window will become too small and restrict the detection capabilities of CSX. For example, if the window contains

Option	Default value	Description
`spx.rt.nr_threads`	#cores	Number of threads.
`spx.rt.cpu_affinity`	0...#cores-1	Thread affinity.
`spx.preproc.xform`	**all** h, v, d br, bc, none	Substructure types that will be detected.
`spx.preproc.heuristic`	ratio (SMP) cost (NUMA)	Heuristic that will try to maximize the compression ratio or minimize the computational cost.
`spx.preproc.sampling`	**none** portion window	Use of statistical sampling.
`spx.preproc.sampling.nr_samples`	10	Number of sampling windows per thread.
`spx.preproc.sampling.portion`	0.01	Portion of the matrix that will be sampled.
`spx.matrix.symmetric`	false	Use the symmetric variant of CSX.
`spx.matrix.split_blocks`	true	Use of the split blocks optimization that improves the evaluation of block types during the selection phase.
`spx.matrix.full_colind`	true (NUMA) false (SMP)	Use of variable sized integers for the column index of each substructure.
`spx.matrix.min_unit_size`	4	Minimum number of nonzero elements required for a substructure unti to be valid.
`spx.matrix.max_unit_size`	255	Maximum number of nonzero elements required for a substructure unit to be valid.
`spx.matrix.min_coverage`	0.1	Minimum percentage of matrix coverage required for a substructure instantiation not to be filtered out of the selection phase.

Table 3.2: Available options for configuring the preprocessing phase of CSX.

only a single row then only horizontal and delta units can be detected. Thus, when the number of participating threads is large, the user is advised to set a smaller number of samples, especially in case of small matrices. The user might of course achieve a similar effect by increasing the sampling portion. We plan to address this issue by further refining the heuristic employed for computing the window size.

From the last group of options, the most important for the user is the `spx.matrix.symmetric` option, which enables the use of the symmetric variant of the CSX format. The rest of the options are configured by default according to a 'best practice' policy, which also takes into account the underlying architecture.

3.2.3 Changing matrix nonzero values

The nonzero pattern of the input matrix fixes the nonzero pattern of the `spx_matrix_t` handle, i.e., the matrix in the CSX format. This means that insertion and deletion of elements is not supported once the matrix has been stored in the CSX format. Modifying existing nonzero values, however, is allowed through the `spx_mat_set_entry()` routine. The `spx_mat_get_entry()` routine is also available for retrieving/obtaining a single value. Both routines assume zero-base indexing in the supplied coordinates of the matrix element by default, but one-based indexing is also supported by providing the optional argument `SPX_INDEXING_ONE_BASED`.

If the input matrix contains explicit zeros, the library treats them as "logical nonzeros" whose values can be modified later.

3.2.4 Storing and retrieving CSX

Although significantly optimized, the preprocessing cost of CSX is still non-negligible. This motivated us to implement an I/O feature that will allow the user to avoid tuning a matrix that has already been converted to the CSX format in a previous session. This feature involves serializing the tuned matrix handle and storing it in a binary file, so it can be used in a future session to reconstruct an equivalent handle and directly perform the SpMV kernel. This achieves a considerable speedup over the optimized preprocessing phase of CSX. Our user API provides the `spx_mat_save()` routine for storing the matrix and the `spx_mat_restore()` routine for loading it from the binary file.

3.2.5 Creating and modifying vector objects

Dense vector objects of type `spx_vector_t` can be either wrappers of user-defined arrays or they can be created and initialized explicitly by the user. In a multithreaded scenario, a vector might need to be partitioned among threads. This is done transparently during the creation process and, thus, cannot be controlled by the user. The

only responsibility of the user is to provide the split points to the vector creation routine through an object of type `spx_partition_t`. This object can be constructed either implicitly during the tuning phase of CSX, and subsequently extracted with the `spx_mat_get_partition()` routine, or explicitly through the `spx_partition_csr()` routine when loading a matrix from the CSR format. In a typical scenario, the user loads the input matrix, converts it to the CSX format (in this phase the split points are automatically computed) and then creates the necessary vector objects as in the following sequence:

```
...
spx_input_t *input = spx_input_load_mmf(file);
spx_matrix_t *A = spx_mat_tune(input);
spx_partition_t *parts = spx_mat_get_partition(A);
spx_vector_t *x = spx_vec_create(size, parts);
...
```

When the vector is created as a wrapper of a user-defined array with `spx_vec_create_from_buff()` and the `SPX_VEC_TUNE` option has been set, the library may select to optimize data allocation and return a tuned array for further use. Example 4.4 in Chapter 4 demonstrates use of tuned vectors.

Table 3.3 summarizes the available routines for creating and modifying vector objects, while table 3.4 shows the routines involved in partitioning.

3.2.6 Reordering

When the matrix structure is very irregular, resulting in an equally irregular access pattern in the right-hand side vector, a significant amount of cache misses is introduced. Additionally, when the matrix suffers from an heterogeneous sparsity pattern a varying flop:byte ratio is exhibited among the participating threads, due to fluctuations in the nonzero density across the matrix, leading to significant load imbalances. Reordering the matrix using a bandwidth-reduction technique has been proposed as a solution to the aforementioned problems. This involves applying row and column permutations in order to bring the nonzero elements of the matrix as close as possible to the main diagonal. This is beneficial to a typical SpMV implementation, since the homogenization of the nonzero element distribution leads to a better access pattern and load balance. For the symmetric version of the kernel (using SparseX with the `spx.matrix.symmetric` option enabled), the obvious effect of this nonzero rearrangement in a mutlithreaded execution is the minimization of the reduction phase overhead.

Our user API currently provides an option for applying the *Reverse Cuthill McKee (RCM)* reordering algorithm that has been proposed for structurally symmetric matrices [Cuthill and McKee, 1969]. Reordering is performed by providing the optional argument `SPX_MAT_REORDER` to the `spx_mat_tune()` routine. The `spx_mat_get_perm()` and `spx_vec_reorder()` routines allow the user to extract and apply the correspond-

Routine	Description
`spx_vec_create`	Creates a vector object.
`spx_vec_create_from_buff`	Creates a vector object as a wrapper of a user-defined array. The input array can be either left unmodified or tuned (SPX_VEC_AS_IS/ SPX_VEC_TUNED).
`spx_vec_create_random`	Creates a randomly filled vector object.
`spx_vec_init`	Initializes the vector object with a fixed value.
`spx_vec_init_rand_range`	Initializes the vector object with randomly generated values.
`spx_vec_get_entry`	Retrieves the value of a single entry.
`spx_vec_set_entry`	Sets the value of a single entry.
`spx_vec_destroy`	Destroys a vector object.

Table 3.3: Available routines for creating and modifying vector objects of type `spx_vector_t`.

Routine	Description
`spx_mat_get_partition`	Retrieves a partitioning object from a tuned matrix.
`spx_partition_csr`	Creates a partitioning type object from a matrix in the CSR format.
`spx_partition_destroy`	Destroys a partitioning object.

Table 3.4: Available routines for handling partitioning objects of type `spx_partition_t`.

Routine	Description
`spx_mat_get_perm`	Retrieves the permutation applied to the matrix after applying the RCM algorithm.
`spx_vec_reorder`	Reorders the vector according to the supplied permutation.
`spx_vec_inv_reorder`	Inverse-reorders a permuted vector according to the supplied permutation

Table 3.5: Routines involved in reordering.

ing permutation on vector objects. For an example that illustrates the use of reordering, see Example 4.3 in Chapter 4.

3.3 Computational routines

This module includes three variations of the SpMV kernel (see Table 3.6) and some vector operations (see Table 3.7). The `spx_matvec_mult()` routine implements a simple multiplication ($y \leftarrow alpha \cdot A \cdot x$), while a more generic version of the kernel is also provided, i.e. $y \leftarrow alpha \cdot A \cdot x + beta \cdot y$, through the `spx_matvec_kernel()` routine. This routine is equivalent to the BLAS Level 2 routine for matrix-by-vector multiplication and follows the BLAS convention in parameter ordering. The last routine in Table 3.6 forms a higher-level implementation of the kernel that "hides" the preprocessing phase of CSX by accepting a matrix in the CSR format as input. This last routine can be efficiently used in a loop, since only the first call will convert the matrix into the CSX format and every subsequent call will use the previously tuned matrix handle. Refer to Example 4.2 of the following chapter for efficient use of this routine.

A wide variety of vector operations is also provided, including addition, subtraction, multiplication and scaling. We must note here that every routine in Table 3.7 is also available in a version that operates on a specific range of the input vectors. These variants could be useful in a multithreaded scenario. For a complete reference of the available vector operations see Appendix A.

3.4 Timing routines

Our interface also provides basic utilities for timing your code. A timer in SparseX is represented by the `spx_timer_t` handle type and can be manipulated with one of the routines in Table 3.8.

Routine	Description
`spx_matvec_mult`	Performs the SpMV kernel $y \leftarrow alpha \cdot A \cdot x$ with a tuned matrix handle as input.
`spx_matvec_kernel`	Performs the kernel $y \leftarrow alpha \cdot A \cdot x + beta \cdot y$ with a tuned matrix handle as input.
`spx_matvec_kernel_csr`	Performs the kernel $y \leftarrow alpha \cdot A \cdot x + beta \cdot y$ with a matrix in the CSR format as input.

Table 3.6: Available routines for sparse matrix-by-vector multiply.

Routine	Description
`spx_vec_add`	Adds two vectors.
`spx_vec_sub`	Subtracts two vectors.
`spx_vec_mul`	Returns the product of two vectors.
`spx_vec_scale`	Scales the input vector by a constant value.
`spx_vec_scale_add`	Scales the input vector by a constant value.

Table 3.7: Available vector operations.

Routine	Description
`spx_timer_start`	Starts the timer.
`spx_timer_pause`	Pauses the timer.
`spx_timer_clear`	Re-initializes the timer.
`spx_timer_get_secs`	Returns the elapsed time in seconds.

Table 3.8: Timing utilities.

3.5 Logging routines

Logging is a critical technique for troubleshooting and maintaining software systems. The logging framework of SparseX was designed to be typesafe, threadsafe (at line level), flexible and as light-weight as possible. It adopts the look-and-feel of C++ streams and can be enabled or disabled both at compile time and runtime.

By default, only the `Error` and `Warning` logging levels are enabled. However, the user may find the `Info` level particularly helpful in understanding the preprocessing phase of CSX while evaluating program performance. For example, `Info` activates the printing of statistics about the detected and encoded substructures during the conversion to the CSX format. Additionally, in case of NUMA architectures, where the performance of the SpMV kernel is sensitive to the correct placement of the involved data on the system's memory nodes, information is given on the success or failure of the corresponding memory allocations. Table 3.9 gives an overview of the available routines for configuring the logging process.

3.6 Error handling in SparseX

In general, the SparseX interface distinguishes three types of errors: (a) fatal errors generated by the operating system, (b) logical errors (i.e., created by the user) that are fatal to the program execution and (c) logical errors that do not lead to program failure The first category may include memory- and file-related errors. The second category may include errors due to invalid arguments supplied to one of the interface's routines. Finally, non-fatal logical errors may occur, for example, when trying to set an option to an invalid value.

The interface handles errors with the use of error handling routines and error codes or invalid handle returns. When an error condition is detected within a SparseX routine, it is treated as following:

- The routine calls the current error handler.
- Regardless of what action the error handler performs, the routine returns an error code or an invalid handle depending on the routine.

The default error handler uses the logging framework described in the previous section to output messages of the following form

[*prefix*]: *message* [*sourcefile*:*lineno*:*function*]

where *prefix* can be either `ERROR` or `WARNING` depending on the error type. When an error of the first category occurs, the default error handler also outputs the error message generated by the operating system and subsequently exits the program with a nonzero exit code. In case of logical errors, on the other hand, returning error codes instead of exiting seemed more appropriate, since most routines make requests on available resources and their failure needs to be recoverable.

Routine	Description
`spx_log_all_console`	Activates all logging levels and redirects output to stderr.
`spx_log_all_file`	Activates all logging levels and redirects output to a logfile.
`spx_log_set_file`	Sets a logging file.
`spx_log_disable_all`	Disables logging of all levels.
`spx_log_error_console`	Activates logging of the `Error` level on stderr.
`spx_log_error_file`	Activates logging of the `Error` level on a file.
`spx_log_warning_console`	Activates logging of the `Warning` level on stderr.
`spx_log_warning_file`	Activates logging of the `Warning` level on a file.
`spx_log_info_console`	Activates logging of the `Info` level on stderr.
`spx_log_info_file`	Activates logging of the `Info` level on a file.
`spx_log_debug_console`	Activates logging of the `Debug` level on stderr.
`spx_log_debug_file`	Activates logging of the `Debug` level on a file.

Table 3.9: Available routines to control logging.

If the user wishes to handle errors in a different manner, she may set her own error handling routine by calling `spx_err_set_handler()`, as long as it conforms to the signature set by our interface. For more details see Appendix A.

A couple of macros are used to make the error handling a bit more convenient. These macros are used throughout the interface and can be employed by the application programmer as well. When an error is first detected, one should set it by calling

`SETERROR_0(error_code)` or
`SETERROR_1(error_code, message)`

Both macros call the current error handler, while the second also supplies a custom message.

4

Examples

This chapter introduces the SparseX interface through a series of examples. Every example highlights different aspects of our API in order to familiarize the user with most of the available routines. These code examples can be located in the `src/examples` subdirectory of the installation directory. They could be used to determine:

- Whether SparseX is working on your system
- How you should call the library
- How to link the library

4.1 Example 1

In our first minimal example the input matrix is loaded from a file in the MMF format. Depending on the number of available cores of the system, serial or multithreaded execution is automatically selected. Random vectors are created and, finally, a loop of 128 iterations of the SpMV kernel is executed and timed through the available timing routines of our interface. The user must not forget to cleanup all objects associated to our interface and shutdown the library before exiting. It is also important to note that vector objects must be created after the tuning has taken place, after retrieving the partitioning handle through the **`spx_mat_get_parts()`** routine.

```
1  /* Initialize library */
2  spx_init();

4  /* Load matrix from MMF file */
5  spx_input_t *input = spx_input_load_mmf(argv[1]);

7  /* Transform to CSX */
8  spx_matrix_t *A = spx_mat_tune(input);

10 /* Create random x and y vectors */
11 spx_partition_t *parts = spx_mat_get_partition(A);
12 spx_vector_t *x = spx_vec_create_random(spx_mat_get_ncols(A),
```

```
13                                                  parts);
14  spx_vector_t *y = spx_vec_create_random(spx_mat_get_nrows(A),
15                                                  parts);

17  /* Run 128 loops of the SpMV kernel */
18  spx_timer_t t;
19  double elapsed_time;
20  int i;

22  spx_timer_clear(&t);
23  spx_timer_start(&t);
24  for (i = 0; i < 128; i++) {
25      spx_matvec_kernel(alpha, A, x, beta, y);
26  }
27  spx_timer_pause(&t);
28  elapsed_time = spx_timer_get_secs(&t);
29  printf("Elapsed time: %lf secs\n", elapsed_time);

31  /* Cleanup */
32  spx_input_destroy(input);
33  spx_mat_destroy(A);
34  spx_partition_destroy(parts);
35  spx_vec_destroy(x);
36  spx_vec_destroy(y);

38  /* Shutdown library */
39  spx_finalize();
```

4.2 Example 2

This example illustrates use of the higher-level multiplication routine **`spx_matvec_kernel_csr()`**. In this case, our input will be in the CSR format and the conversion to the CSX format will be hidden in the multiplication routine. First we explicitly partition the matrix in the CSR format, through the **`spx_partition_csr()`** routine and, subsequently, the x and y vector views are created from the user-defined arrays. The **`SPX_VEC_AS_IS`** option indicates that no tuning will be performed to the input array, so the second argument of the routine can be `NULL`. Finally, a loop of 128 iterations of the SpMV kernel is executed and the library objects are destroyed.

```
1  /* Define CSR data structures */
2  spx_index_t rowptr[] = {0,5,6,10,15,18,22,24,29,33,38};
3  spx_index_t colind[] = {0,1,2,3,8,7,0,1,6,9,0,1,3,5,9,0,1,
4                          9,0,1,5,9,2,3,2,3,4,5,7,2,3,4,5,2,
5                          3,4,5,9};
6  spx_value_t values[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,
```

```
7                           15,16,17,18,19,20,21,22,23,24,25,
8                           26,26.1,26.2,27,28,29,29.1,29.2,
9                           30,31,31.1,31.2,32};

11 /* Define vector arrays */
12 spx_value_t x[] = {1,2,3,4,5,6,7,8,9,10};
13 spx_value_t y[] = {0.19,0.28,0.31,0.42,1.32,
14                    2.64,0.75,2.36,0.91,1};

16 spx_index_t nrows, ncols;
17 spx_value_t alpha, beta;

19 nrows = 10; ncols = 10;
20 alpha = 0.42; beta = 0.10;

22 /* Initialize library */
23 spx_init();

25 /* Partition matrix */
26 spx_partition_t *parts = spx_partition_csr(rowptr, nrows,
27                                            nr_parts);

29 /* Create vectors from arrays */
30 spx_vector_t *x_view = spx_vec_create_from_buff(x, NULL,
31                                                 ncols, parts,
32                                                 SPX_VEC_AS_IS);
33 spx_vector_t *y_view = spx_vec_create_from_buff(y, NULL,
34                                                 ncols, parts,
35                                                 SPX_VEC_AS_IS);

37 /* Declare a tuned matrix handle */
38 spx_matrix_t *A = SPX_INVALID_MAT;

40 /* Run 128 iterations of the SpMV kernel:
41        y <- alpha*A*x + beta*y */
42 for (i = 0; i < 128; i++) {
43     spx_matvec_kernel_csr(&A, nrows, ncols, rowptr, colind,
44                           values, alpha, x_view, beta, y_view);
45 }

47 /* Cleanup interface objects */
48 spx_mat_destroy(A);
49 spx_vec_destroy(x_view);
50 spx_vec_destroy(y_view);
51 spx_partition_destroy(parts);

53 /* Shutdown library */
54 spx_finalize();
```

4.3 Example 3

This example illustrates the use of reorderng with SparseX. The matrix in this example is loaded from a file in the MMF format. Multithreaded execution is selected with 2 threads and a cpu affinity of {0,1}. The symmetric variant of CSX is selected and tuning is performed explicitly with sampling enabled, but this time using sampling windows of a fixed size, and detecting only delta units. Reordering is enabled through the **SPX_MAT_REORDER** option of the **spx_mat_tune()** routine. We must observe that the x and y vectors must be explicitly reordered before executing the kernel and inverse-reordered after the execution.

```
1  /* Initialize library */
2  spx_init();

4  /* Load matrix from file */
5  spx_input_t *input = spx_input_load_mmf(file);

7  /* Set tuning options */
8  spx_option_set("spx.rt.nr_threads", "2");
9  spx_option_set("spx.rt.cpu_affinity", "0,1");
10 spx_option_set("spx.matrix.symmetric", "true");
11 spx_option_set("spx.preproc.xform", "none");
12 spx_option_set("spx.preproc.sampling", "window");
13 spx_option_set("spx.preproc.sampling.window_size", "600");

15 /* Transform to CSX */
16 spx_matrix_t *A = spx_mat_tune(input, SPX_MAT_REORDER);

18 /* Create randomly filled vectors */
19 spx_index_t ncols = spx_mat_get_ncols(A);
20 spx_index_t nrows = spx_mat_get_nrows(A);
21 spx_partition_t *parts = spx_mat_get_partition(A);
22 spx_vector_t *x = spx_vec_create_random(ncols, parts);
23 spx_vector_t *y = spx_vec_create_random(nrows, parts);

25 /* Reorder vectors */
26 spx_perm_t *p = spx_mat_get_perm(A);
27 spx_vec_reorder(x, p);
28 spx_vec_reorder(y, p);

30 /* Run the SpMV kernel:
31    y <- alpha*A*x + beta*y */
32 spx_matvec_kernel(alpha, A, x, beta, y);

34 /* Inverse-reorder the output vector */
35 spx_vec_inv_reorder(y, p);
```

```
37 /* Cleanup interface objects */
38 spx_input_destroy(input);
39 spx_mat_destroy(A);
40 spx_vec_destroy(x);
41 spx_vec_destroy(y);
42 spx_partition_destroy(parts);

44 /* Shutdown library */
45 spx_finalize();
```

4.4 Example 4

Our last example illustrates use of the vector tuning option of our library, which can be used to extract higher performance, especially on NUMA platforms. In this case, the user must supply a pointer to the tuned data as the second argument of the **`spx_vec_create_from_buff()`** routine (see **`x_tuned`** and **`y_tuned`** at line 11). If the data is actually tuned by our library, the pointers will point to different memory regions. However, since the data is selectively tuned, the user must always check the pointers for equality. If the pointers differ then the tuned buffer should be used instead of the original from this point on. The common free() function applies to this buffer and will have to be explicitly called by the user.

```
1  /* Initialize library */
2  spx_init();

4  /* Load matrix from file */
5  spx_input_t *input = spx_input_load_mmf(argv[1]);

7  /* Transform to CSX */
8  spx_matrix_t *A = spx_mat_tune(input);

10 /* Create x and y vector views from the corresponding buffers */
11 spx_value_t *x_tuned, *y_tuned;
12 spx_partition_t *parts = spx_mat_get_partition(A);
13 spx_vector_t *x_view = spx_vec_create_from_buff(x, &x_tuned,
14                                                 ncols, parts,
15                                                 SPX_VEC_TUNE);
16 spx_vector_t *y_view = spx_vec_create_from_buff(y, &y_tuned,
17                                                 nrows, parts,
18                                                 SPX_VEC_TUNE);

20 /* Run 128 loops of the SpMV kernel */
21 spx_value_t alpha = 0.8, beta = 0.42;
22 const size_t nr_loops = 128;
```

```
24 for (i = 0; i < nr_loops; i++) {
25     spx_matvec_kernel(alpha, A, x_view, beta, y_view);
26 }

28 /* From this point on the user can use the tuned buffers */
29 if (x_tuned != x) {
30     x = x_tuned;
31 }

33 if (y_tuned != y) {
34     y = y_tuned;
35 }

37 /* ... */
38 /* ... */
39 /* ... */

41 /* Cleanup */
42 spx_input_destroy(input);
43 spx_mat_destroy(A);
44 spx_partition_destroy(parts);
45 spx_vec_destroy(x_view);
46 spx_vec_destroy(y_view);

48 /* The user can apply the free() function on the tuned buffers */
49 free(x);
50 free(y);

52 /* Shutdown library */
53 spx_finalize();
```

A

C bindings reference

Generated by Doxygen 1.8.6

A.1 Available routines

- void spx_init ()
- void spx_finalize ()
- spx_input_t * spx_input_load_csr (spx_index_t *rowptr, spx_index_t *colind, spx_value_t *values, spx_index_t nr_rows, spx_index_t nr_cols,...)
- spx_input_t * spx_input_load_mmf (const char *filename)
- spx_error_t spx_input_destroy (spx_input_t *input)
- spx_matrix_t * spx_mat_tune (spx_input_t *input,...)
- spx_error_t spx_mat_get_entry (const spx_matrix_t *A, spx_index_t row, spx_index_t column, spx_value_t *value,...)
- spx_error_t spx_mat_set_entry (spx_matrix_t *A, spx_index_t row, spx_index_t column, spx_value_t value,...)
- spx_error_t spx_mat_save (const spx_matrix_t *A, const char *filename)
- spx_matrix_t * spx_mat_restore (const char *filename)
- spx_index_t spx_mat_get_nrows (const spx_matrix_t *A)
- spx_index_t spx_mat_get_ncols (const spx_matrix_t *A)
- spx_index_t spx_mat_get_nnz (const spx_matrix_t *A)
- spx_partition_t * spx_mat_get_partition (spx_matrix_t *A)
- spx_perm_t * spx_mat_get_perm (const spx_matrix_t *A)
- spx_error_t spx_matvec_mult (spx_value_t alpha, const spx_matrix_t *A, spx_vector_t *x, spx_vector_t *y)
- spx_error_t spx_matvec_kernel (spx_value_t alpha, const spx_matrix_t *A, spx_vector_t *x, spx_value_t beta, spx_vector_t *y)
- spx_error_t spx_matvec_kernel_csr (spx_matrix_t **A, spx_index_t nr_rows, spx_index_t nr_cols, spx_index_t *rowptr, spx_index_t *colind, spx_value_t *values, spx_value_t alpha, spx_vector_t *x, spx_value_t beta, spx_vector_t *y)
- spx_error_t spx_mat_destroy (spx_matrix_t *A)
- spx_partition_t * spx_partition_csr (spx_index_t *rowptr, spx_index_t nr_rows, size_t nr_threads)
- spx_error_t spx_partition_destroy (spx_partition_t *p)
- void spx_option_set (const char *option, const char *string)
- spx_vector_t * spx_vec_create (size_t size, spx_partition_t *p)
- spx_vector_t * spx_vec_create_from_buff (spx_value_t *buff, spx_value_t *tuned, size_t size, spx_partition_t *p, spx_vecmode_t mode)
- spx_vector_t * spx_vec_create_random (size_t size, spx_partition_t *p)
- void spx_vec_init (spx_vector_t *v, spx_value_t val)
- void spx_vec_init_part (spx_vector_t *v, spx_value_t val, spx_index_t start, spx_index_t end)

- void spx_vec_init_rand_range (spx_vector_t *v, spx_value_t max, spx_value_t min)
- spx_error_t spx_vec_set_entry (spx_vector_t *v, spx_index_t idx, spx_value_t val,...)
- void spx_vec_scale (spx_vector_t *v1, spx_vector_t *v2, spx_value_t num)
- void spx_vec_scale_add (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_value_t num)
- void spx_vec_scale_add_part (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_value_t num, spx_index_t start, spx_index_t end)
- void spx_vec_add (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3)
- void spx_vec_add_part (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_index_t start, spx_index_t end)
- void spx_vec_sub (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3)
- void spx_vec_sub_part (spx_vector_t *v1, spx_vector_t *v2, spx_vector_t *v3, spx_index_t start, spx_index_t end)
- spx_value_t spx_vec_mul (const spx_vector_t *v1, const spx_vector_t *v2)
- spx_value_t spx_vec_mul_part (const spx_vector_t *v1, const spx_vector_t *v2, spx_index_t start, spx_index_t end)
- spx_error_t spx_vec_reorder (spx_vector_t *v, spx_perm_t *p)
- spx_error_t spx_vec_inv_reorder (spx_vector_t *v, spx_perm_t *p)
- void spx_vec_copy (const spx_vector_t *v1, spx_vector_t *v2)
- int spx_vec_compare (const spx_vector_t *v1, const spx_vector_t *v2)
- void spx_vec_print (const spx_vector_t *v)
- void spx_vec_destroy (spx_vector_t *v)
- void spx_timer_clear (spx_timer_t *t)
- void spx_timer_start (spx_timer_t *t)
- void spx_timer_pause (spx_timer_t *t)
- double spx_timer_get_secs (spx_timer_t *t)
- void spx_log_disable_all ()
- void spx_log_error_console ()
- void spx_log_warning_console ()
- void spx_log_info_console ()
- void spx_log_verbose_console ()
- void spx_log_debug_console ()
- void spx_log_error_file ()
- void spx_log_warning_file ()
- void spx_log_info_file ()
- void spx_log_verbose_file ()
- void spx_log_debug_file ()

- void spx_log_all_console ()
- void spx_log_all_file (const char *file)
- void spx_log_set_file (const char *file)
- spx_errhandler_t spx_err_get_handler ()
- void spx_err_set_handler (spx_errhandler_t new_handler)

A.2 Routine documentation

A.2.1 void spx_init ( )

Library initialization routine.

A.2.2 void spx_finalize ( )

Library shutdown routine.

A.2.3 spx_input_t* spx_input_load_csr (spx_index_t * *rowptr,* spx_index_t * *colind,* spx_value_t * *values,* spx_index_t *nr_rows,* spx_index_t *nr_cols,* ...)

Creates and returns a valid tunable matrix object from a Compressed Sparse Row (CSR) representation.

Parameters

in	*rowptr*	array *rowptr* of the CSR format.
in	*colind*	array *colind* of the CSR format.
in	*values*	array *values* of the CSR format.
in	*nr_rows*	number of rows of the matrix.
in	*nr_cols*	number of columns of the matrix.
in	...	argument that specifies the indexing (either SPX_IND↩ EX_ZERO_BASED or SPX_INDEX_ONE_BASED).

Returns

a handle to the input matrix or SPX_INVALID_INPUT in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments are invalid.
2. SPX_ERR_INPUT_MAT: the input data arrays do not correspond to a valid CSR representation.

A.2.4 spx_input_t* spx_input_load_mmf (const char * *filename*)

Creates and returns a valid tunable matrix object from a file in the Matrix Market File Format (MMF).

Parameters

in	*filename*	name of the file where the matrix is stored.

Returns

a handle to the input matrix or SPX_INVALID_INPUT in case an error occurs.

Possible error conditions

1. SPX_ERR_FILE: the file does not exist or an error occured while trying to read it (possibly not in valid MMF format).

A.2.5 spx_error_t spx_input_destroy (spx_input_t * *input*)

Releases any memory internally used by the sparse matrix input handle *input*.

Parameters

in	*input*	the input matrix handle.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the input matrix handle is invalid.

A.2.6 spx_matrix_t* spx_mat_tune (spx_input_t * *input, ...*)

Converts the input matrix into the CSX format by applying all the options previously set with the spx_option_set() routine. In case no options have been explicitly set, the default values are used (see Table 3.2 of the User's Guide).

Parameters

in	*input*	the input matrix handle.
in	...	optional flag that indicates whether the matrix should be reordered by applying the Reverse Cuthill McKee algorithm (SPX_MAT_REORDER).

Returns

a handle to the tuned matrix or SPX_INVALID_MAT in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the input matrix handle is invalid.
2. SPX_ERR_TUNED_MAT: conversion to the CSX format failed.

A.2.7 spx_error_t spx_mat_get_entry (const spx_matrix_t * *A,* spx_index_t *row,* spx_index_t *column,* spx_value_t * *value,* ...)

Returns the value of the corresponding nonzero element in (*row*, *column*), where *row* and *column* can be either zero- or one-based indexes. Default indexing is zero-based, but it can be overidden through the optional flag. If the element exists, its value is returned in *value*.

Parameters

in	*A*	the tuned matrix handle.
in	*row*	the a row of the element to be retrieved.
in	*column*	the column of the element to be retrieved.
out	*value*	the value of the element in (*row*, *column*).
in	...	argument that specifies the indexing (either SPX_IND↩ EX_ZERO_BASED or SPX_INDEX_ONE_BASED).

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_OUT_OF_BOUNDS: the element is out of range.
3. SPX_ERR_ENTRY_NOT_FOUND: the element doesn't exist (i.e. is not nonzero).

A.2.8 spx_error_t spx_mat_set_entry (spx_matrix_t * *A,* spx_index_t *row,* spx_index_t *column,* spx_value_t *value,* ...)

Sets the value of the corresponding element in (*row*, *column*), where *row* and *column* can be either zero- or one-based indexes. Default indexing is zero-based, but it can be overidden through the optional flag.

Parameters

in	*A*	the tuned matrix handle.
in	*row*	the row of the element to be set.
in	*column*	the column of the element to be set.
in	*value*	the new value of the element in (*row*, *column*).
in	...	argument that specifies the indexing (either SPX_IND↩ EX_ZERO_BASED or SPX_INDEX_ONE_BASED).

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_OUT_OF_BOUNDS: the element is out of range.
3. SPX_ERR_ENTRY_NOT_FOUND: the element doesn't exist (i.e. is not nonzero).

A.2.9 spx_error_t spx_mat_save (const spx_matrix_t * *A*, const char * *filename*)

Stores the matrix in the CSX format into a binary file.

Parameters

in	*A*	the tuned matrix handle.
in	*filename*	the name of the binary file where the matrix will be dumped.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

Possible warnings

1. SPX_WARN_CSXFILE: a filename hasn't been supplied so the default *csx_file.bin* will be used.

A.2.10 spx_matrix_t* spx_mat_restore (const char * *filename*)

Reconstructs the matrix in the CSX format from a binary file.

Parameters

in	*filename*	the name of the file where the matrix is stored.

Returns

a handle to the tuned matrix or SPX_INVALID_MAT in case an error occurs.

Possible error conditions

1. SPX_ERR_FILE: invalid filename argument or file read error.
2. SPX_ERR_TUNED_MAT: reproducing the matrix failed.

A.2.11 spx_index_t spx_mat_get_nrows (const spx_matrix_t * *A*)

Returns the number of rows of the matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

the number of rows.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

A.2.12 spx_index_t spx_mat_get_ncols (const spx_matrix_t * *A*)

Returns the number of columns of the matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

the number of columns.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

A.2.13 spx_index_t spx_mat_get_nnz (const spx_matrix_t * *A*)

Returns the number of nonzero elements of the matrix.

Parameters

in	*A*	the tuned matrix handle.

Returns

the number of nonzeros.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.

A.2.14 spx_partition_t* spx_mat_get_partition (spx_matrix_t * *A*)

Returns a partitioning object for the given matrix.

Parameters

in	A	the tuned matrix handle.

Returns

a valid partitioning object or SPX_INVALID_PART in case an error occurs.

A.2.15 spx_perm_t* spx_mat_get_perm (const spx_matrix_t * *A*)

Returns the permutation computed for the supplied matrix by applying the Reverse Cuthill-McKee algorithm.

Parameters

in	A	the tuned matrix handle.

Returns

a handle to the permutation object or SPX_INVALID_PERM in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid or the matrix has not been reordered.

A.2.16 spx_error_t spx_matvec_mult (spx_value_t *alpha,* const spx_matrix_t * *A,* spx_vector_t * *x,* spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x \tag{A.1}$$

where *alpha* is a scalar, x and y are vectors and A is a sparse matrix in the CSX format.

Parameters

in	A	the tuned matrix handle.
in	*alpha*	a scalar.
in	x	the input vector.
in,out	y	the output vector.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.17 spx_error_t spx_matvec_mult (spx_value_t *alpha,* const spx_matrix_t * *A,* spx_vector_t * *x,* spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x \tag{A.2}$$

where *alpha* is a scalar, *x* and *y* are vectors and *A* is a sparse matrix in the CSX format.

Parameters

in	*A*	the tuned matrix handle.
in	*alpha*	a scalar.
in	*x*	the input vector.
in, out	*y*	the output vector.

Returns

an error code.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.18 spx_error_t spx_matvec_kernel (spx_value_t *alpha,* const spx_matrix_t * *A,* spx_vector_t * *x,* spx_value_t *beta,* spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x + beta \cdot y \tag{A.3}$$

where *alpha* and *beta* are scalars, *x* and *y* are vectors and *A* is a sparse matrix in the CSX format.

Parameters

in	*A*	the tuned matrix handle.
in	*alpha*	a scalar.
in	*x*	the input vector.
in	*beta*	a scalar.
in, out	*y*	the output vector.

Returns

an error code.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.19 spx_error_t spx_matvec_kernel_csr (spx_matrix_t ** *A*, spx_index_t *nr_rows*, spx_index_t *nr_cols*, spx_index_t * *rowptr*, spx_index_t * *colind*, spx_value_t * *values*, spx_value_t *alpha*, spx_vector_t * *x*, spx_value_t *beta*, spx_vector_t * *y*)

Performs a matrix-vector multiplication of the following form:

$$y = alpha \cdot A \cdot x + beta \cdot y \tag{A.4}$$

where *alpha* and *beta* are scalars, x and y are vectors and A is a sparse matrix. The matrix is originally given in the CSR format and converted internally into the CSX format. This higher-level routine hides the preprocessing phase of CSX. It can be efficiently used in a loop, since only the first call will convert the matrix into the CSX format and every subsequent call will use the previously tuned matrix handle.

Parameters

in	*A*	either a pointer to an invalid matrix handle or a tuned matrix handle. If (*A) is equal to SPX_INVALID_MAT then the matrix in the CSR format is first converted to CSX. Otherwise, the valid (previously) tuned matrix handle is used to perform the multiplication.
in	*nr_rows*	number of rows of the matrix *A*.
in	*nr_cols*	number of columns of the matrix *A*.
in	*rowptr*	array *rowptr* of the CSR format.
in	*colind*	array *colind* of the CSR format.
in	*values*	array *values* of the CSR format.
in	*alpha*	a scalar.
in	*x*	the input vector.
in	*beta*	a scalar.
in,out	*y*	the output vector.

Returns

an error code.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.
2. SPX_ERR_INPUT_MAT: the input data arrays do not correspond to a valid CSR representation.
3. SPX_ERR_DIM: matrix and vector dimensions are not compatible.

A.2.20 spx_error_t spx_mat_destroy (spx_matrix_t * *A*)

Releases any memory internally used by the tuned matrix handle *A*.

Parameters

in	*A*	the tuned matrix handle.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the matrix handle is invalid.
2. SPX_ERR_MEM_FREE: deallocation failed.

A.2.21 spx_partition_t* spx_partition_csr (spx_index_t * *rowptr,* spx_index_t *nr_rows,* size_t *nr_threads*)

Creates a partitioning object for the matrix in the Compressed Sparse Row (CSR) format. This routine should be used in conjunction with the spx_matvec_kernel_csr() multiplication routine.

Parameters

in	*rowptr*	array *rowptr* of the CSR format.
in	*nr_rows*	number of rows of the matrix.
in	*nr_threads*	number of partitions of the matrix.

Returns

a partitioning object or SPX_INVALID_PART in case an error occurs.

A.2.22 spx_error_t spx_partition_destroy (spx_partition_t * *p*)

Releases any memory internally used by the partitioning handle *p*.

Parameters

in	*p*	the partitioning handle.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: the partitioning handle is invalid.
2. SPX_ERR_MEM_FREE: deallocation failed.

A.2.23 spx_error_t spx_option_set (const char * *option,* const char * *string*)

Sets the *option* according to the description in *string* for the tuning process to follow. For available tuning options and how to set them refer to Table 3.2 of the User's Guide.

Parameters

in	*option*	the option to be set.
in	*string*	a description of how to set the option.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_MEM_FREE: deallocation failed.

A.2.24 spx_vector_t* spx_vec_create (size_t *size,* spx_partition_t * *p*)

Creates and returns a vector object, whose values must be explicitly initialized.

Parameters

in	*size*	the size of the vector to be created.
in	*p*	a partitioning handle.

Returns

a valid vector object or SPX_INVALID_VEC in case of an error.

A.2.25 spx_vector_t* spx_vec_create_from_buff (spx_value_t * *buff,* spx_value_t * *tuned,* size_t *size,* spx_partition_t * *p,* spx_vecmode_t *mode*)

Creates and returns a valid vector object, whose values are mapped to a user-defined array. If SPX_VEC_AS_IS is set, then the input buffer will be shared with the library and modifications will directly apply to it. In this case pointers *buff* and *tuned* point to the same memory location. If SPX_VEC_TUNE is selected, the buffer provided by the user might be copied into an optimally allocated buffer (depending on the platform) and *tuned* might point to this buffer. Thus, the user must always check whether *buff* equals *tuned.* If the buffer is actually tuned, then it should be used instead of the original. The common free() function applies to this buffer and will have to be explicitly called by the user.

Parameters

in	*buff*	the user-supplied buffer.
in	*tuned*	the tuned buffer.
in	*size*	the size of the buffer.
in	*p*	a partitioning handle.
in	*mode*	the vector mode (either SPX_VEC_AS_IS or SPX_↩ VEC_TUNE).

Returns

a valid vector object or SPX_INVALID_VEC in case of an error.

A.2.26 spx_vector_t* spx_vec_create_random (size_t *size*, spx_partition_t * *p*)

Creates and returns a vector object, whose values are randomly filled.

Parameters

in	*size*	the size of the vector to be created.
in	*p*	a partitioning handle.

Returns

a valid vector object or SPX_INVALID_VEC in case of an error.

A.2.27 void spx_vec_init (spx_vector_t * *v*, spx_value_t *val*)

Initializes the vector object *v* with *val*.

Parameters

in	*v*	a valid vector object.
in	*val*	the value to fill the vector with.

A.2.28 void spx_vec_init_part (spx_vector_t * *v*, spx_value_t *val*, spx_index_t *start*, spx_index_t *end*)

Initializes the [*start*:*end*] range of the vector object *v* with *val*.

Parameters

in	*v*	a valid vector object.

in	*val*	the value to fill the vector with.
in	*start*	starting index.
in	*end*	ending index.

A.2.29 void spx_vec_init_rand_range (spx_vector_t * *v*, spx_value_t *max*, spx_value_t *min*)

Initializes the vector object *v* with random values in the range [*min*, *max*].

Parameters

in	*v*	a valid vector object.
in	*max*	maximum value of initializing range.
in	*min*	minimum value of initializing range.

A.2.30 spx_error_t spx_vec_set_entry (spx_vector_t * *v*, spx_index_t *idx*, spx_value_t *val*, ...)

Sets the element at index *idx* of vector *v* to be equal to *val*. Default indexing is zero-based, but it can be overidden through the optional flag.

Parameters

in	*v*	a valid vector object.
in	*idx*	an index inside the vector.
in	*val*	the value to be set.
in	...	argument that specifies the indexing (either SPX_IND↩ EX_ZERO_BASED or SPX_INDEX_ONE_BASED).

A.2.31 void spx_vec_scale (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_value_t *num*)

Scales the input vector *v1* by a constant value *num* and places the result in vector *v2*, i.e. $v2 = num * v1$.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*num*	the constant by which to scale *v1*.

A.2.32 void spx_vec_scale_add (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_value_t *num*)

Scales the input vector *v2* by a constant value *num*, adds the result to vector *v1* and places the result in vector *v3*, i.e. $v3 = v1 + num * v2$.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*num*	the scalar by which to scale *v1*.

A.2.33 void spx_vec_scale_add_part (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_value_t *num*, spx_index_t *start*, spx_index_t *end*)

v3[*start*:*end*] = *v1*[*start*:*end*] + *num* * *v2*[*start*:*end*]

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*num*	the scalar by which to scale *v1*.
in	*start*	starting index.
in	*end*	ending index.

A.2.34 void spx_vec_add (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*)

Adds the input vectors *v1* and *v2* and places the result in *v3*, i.e. *v3* = *v1* + *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.

A.2.35 void spx_vec_add_part (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_index_t *start*, spx_index_t *end*)

Adds the range [*start*:*end*] of the input vectors *v1* and *v2* and places the result in *v3*, i.e. *v3*[*start*:*end*] = *v1*[*start*:*end*] + *v2*[*start*:*end*].

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

in	*v3*	a valid vector object.
in	*start*	starting index.
in	*end*	ending index.

A.2.36 void spx_vec_sub (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*)

Subtracts the input vector *v2* from *v1* and places the result in *v3*, i.e. *v3* = *v1* - *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.

A.2.37 void spx_vec_sub_part (spx_vector_t * *v1*, spx_vector_t * *v2*, spx_vector_t * *v3*, spx_index_t *start*, spx_index_t *end*)

Subtracts the input vector *v2* from *v1* in the range [*start*:*end*] and places the result in *v3*, i.e. *v3*[*start*:*end*] = *v1*[*start*:*end*] - *v2*[*start*:*end*].

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*v3*	a valid vector object.
in	*start*	starting index.
in	*end*	ending index.

A.2.38 spx_value_t spx_vec_mul (const spx_vector_t * *v1*, const spx_vector_t * *v2*)

Returns the product of the input vectors *v1* and *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

Returns

the product of the input vectors.

A.2.39 spx_value_t spx_vec_mul_part (const spx_vector_t * *v1*, const spx_vector_t * *v2*, spx_index_t *start*, spx_index_t *end*)

Returns the product of the range [*start*:*end*] of the input vectors *v1* and *v2*.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.
in	*start*	starting index.
in	*end*	ending index.

Returns

the product of the input vectors.

A.2.40 spx_error_t spx_vec_reorder (spx_vector_t * *v*, spx_perm_t * *p*)

Reorders the input vector v according to the permutation p, leaving the original vector intact.

Parameters

in	*v*	a valid vector object.
in	*p*	a permutation.

Returns

the permuted input vector or SPX_INVALID_VEC in case an error occurs.

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.

A.2.41 spx_error_t spx_vec_inv_reorder (spx_vector_t * *v*, spx_perm_t * *p*)

Inverse-reorders the permuted input vector *v*, according to the permutation *p*.

Parameters

in, out	*v*	the vector object to be inverse-reordered.
in	*p*	a permutation.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

Possible error conditions

1. SPX_ERR_ARG_INVALID: any of the input arguments is invalid.

A.2.42 void spx_vec_copy (const spx_vector_t * *v1*, spx_vector_t * *v2*)

Copies the values of v1 to v2.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

A.2.43 int spx_vec_compare (const spx_vector_t * *v1*, const spx_vector_t * *v2*)

Compares the values of v1 and v2. If they are equal it returns 0, else -1.

Parameters

in	*v1*	a valid vector object.
in	*v2*	a valid vector object.

A.2.44 void spx_vec_print (const spx_vector_t * *v*)

Prints the input vector v.

Parameters

in	*v*	a valid vector object.

A.2.45 spx_error_t spx_vec_destroy (spx_vector_t * *v*)

Destroys the input vector v.

Parameters

in	*v*	a valid vector object.

Returns

an error code (SPX_SUCCESS or SPX_FAILURE).

A.2.46 void spx_timer_clear (spx_timer_t * *t*)

Initialises a timer object.

Parameters

in	*t*	timer object to be initialized.

A.2.47 void spx_timer_start (spx_timer_t * *t*)

Starts a timer object.

Parameters

in	*t*	timer object to be launched.

A.2.48 void spx_timer_pause (spx_timer_t * *t*)

Pauses a timer object.

Parameters

in	*t*	timer object to be paused.

A.2.49 double spx_timer_get_secs (spx_timer_t * *t*)

Returns the elapsed time in seconds.

Parameters

in	*t*	a timer object.

Returns

the elapsed seconds.

A.2.50 void spx_log_disable_all ( )

Disables logging in SparseX.

A.2.51 void spx_log_error_console ( )

Activates logging of the Error level on stderr.

A.2.52 void spx_log_warning_console ( )

Activates logging of the Warning level on stderr.

A.2.53 void spx_log_info_console ( )

Activates logging of the Info level on stderr.

A.2.54 void spx_log_verbose_console ( )

Activates logging of the Verbose level on stderr.

A.2.55 void spx_log_debug_console ( )

Activates logging of the Debug level on stderr.

A.2.56 void spx_log_error_file ( )

Activates logging of the Error level on a file. The file name should be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.57 void spx_log_warning_file ( )

Activates logging of the Warning level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.58 void spx_log_info_file ( )

Activates logging of the Info level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.59 void spx_log_verbose_file ( )

Activates logging of the Verbose level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.60 void spx_log_debug_file ( )

Activates logging of the Debug level on a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created.

A.2.61 void spx_log_all_console ( )

Activates all logging levels and redirects output to stderr.

A.2.62 void spx_log_all_file (const char * *file*)

Activates all logging levels and redirects output to a file. The file name must be previously provided through the spx_log_set_file() routine. If not, a default "sparsex.log" file will be created. If the file already exists it will be overwritten.

Parameters

in	*file*	a filename.

A.2.63 void spx_log_set_file (const char * *file*)

Sets the file that will be used when logging is redirected to a file. If the file already exists it will be overwritten.

Parameters

in	*file*	a filename.

A.2.64 spx_errhandler_t spx_err_get_handler ( )

Returns a pointer to the current error handler (either default or user-defined).

Returns

the current error handling routine.

A.2.65 void spx_err_set_handler (spx_errhandler_t *handler*)

This function allows the user to change the default error handling policy with a new one, which must conform to the signature provided above.

Parameters

in	*handler*	user-defined routine.

B

Acknowledgements

The SparseX library was developed in the Computing Systems Laboratory of the National Technical University of Athens (NTUA) and is actively maintained by:

- **Vasileios Karakasis** <bkk@cslab.ece.ntua.gr>
- **Athena Elafrou** <athena@cslab.ece.ntua.gr>

Past contributors include:

- **Kornilios Kourtis** <kkourt@cslab.ece.ntua.gr>
- **Thodoris Gountouvas** <thgoud@cslab.ece.ntua.gr>

Bibliography

J. Ankit. pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures. 2008.

S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013. URL http://www.mcs.anl.gov/petsc.

R. Boisvert, R Pozo, and Karin Remington. The matrix market exchange formats: Initial design. Technical Report NISTIR-5935, National Institute of Standards and Technology, December 1996.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*. ACM, 1969.

A. Elafrou. Sparsex: A library for the optimization of the spmv kernel on multicore architectures. Master's thesis, National Technical University of Athens, 2013.

Intel® Coorporation. *Intel® Math Kernel Library*. Intel® Coorporation, 2013. URL http://software.intel.com/en-us/intel-mkl.

V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(10):1930–1940, 2013. IEEE.

K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. CSX: An extended compression format for SpMV on shared memory systems. In *16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, San Antonio, TX, USA, 2011. ACM.

R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. volume 16, 2005.