An interface for a self-adapting sparse matrix kernel library

June 3, 2004

Abstract

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

List of Symbols 3
List of Tables 3
List of Listings 3

1 Goals and Motivation 5

2 An Introduction to the Tuning Interface by Example 7
   2.1 Basic usage: gradual migrating applications 7
   2.2 Providing explicit tuning hints 10
   2.3 Tuning based on implicit profiling 12

3 Interface 12
   3.1 Basic scalar types 14
   3.2 Creating and modifying matrix and vector objects 14
      3.2.1 Creating matrix objects 14
      3.2.2 Changing matrix non-zero values 16
      3.2.3 Vector objects 17
   3.3 Executing kernels 18
      3.3.1 Applying the transpose of a matrix 18
      3.3.2 Aliasing 18
      3.3.3 Scalars vs. 1x1 matrix objects 20
      3.3.4 Compatible dimensions for matrix multiplication 20
      3.3.5 Floating point exceptions 20
   3.4 Tuning 20
      3.4.1 Providing workload hints explicitly 21
      3.4.2 Providing structural hints 23
      3.4.3 Initiating tuning 24
      3.4.4 Accessing the permuted form 24
   3.5 Saving and restoring tuning transformations 25
   3.6 Handling errors 29

4 Example: Biconjugate Gradients 30

5 A Tuning Transformation Language 32
   5.1 Basic transformations 32
   5.2 A complex splitting example 32
   5.3 Example of reordering and splitting 33
   5.4 Switch-to-dense example 34

6 Approaches that Complement Libraries 34

References 36
LIST OF SYMBOLS

A  Valid input matrix representations  40

B  Bindings Reference  42
  B.1 Matrix object creation and modification  42
  B.2 Vector object creation  54
  B.3 Kernels  57
  B.4 Tuning  62
  B.5 Permutations  67
  B.6 Saving and restoring tuning transformations  69
  B.7 Error handling  70

List of Symbols

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

List of Tables

1  Creating and modifying matrix and vector objects  15
2  Input matrix properties  17
3  Sparse kernels  19
4  Matrix transpose options (type bebop_matop_t)  19
5  Matrix-transpose-times-matrix options (type bebop_ataop_t)  19
6  Tuning primitives  21
7  Available structural hints (type bebop_tunehint_t)  22
8  Symbolic calling frequency constants (type int)  23
9  Symbolic vector views for workload hints (type bebop_vecview_t)  23
10  Tuning status codes  24
11  Extracting and applying permuted forms  25
12  Saving and restoring tuning transformations  25
13  Error handling routines  30
List of Listings

1  A usage example without tuning ........................................ 8
2  An example of basic explicit tuning ............................... 11
3  An example of implicit tuning ........................................ 13
4  An example of extracting permutations ........................ 26
5  An example of saving transformations ......................... 27
6  An example of applying transformations ...................... 28
1 Goals and Motivation

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

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

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

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

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

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

3. **Offer higher-level memory hierarchy-friendly kernels**: The particular kernels defined in our interface are a superset of those available in similar library interfaces, including the recent Sparse BLAS standard [16, 17] and the SPARSKIT library [39], among others [19]. These additional “higher-level” kernels have inherently more opportunities for reuse.
For example, in addition to the SpMV operation \( y \leftarrow A \cdot x \), we include the kernel \( y \leftarrow A^T A \cdot x \) in which \( A \) may be read from main memory only once. Compared to a register-blocked two-step implementation, \( t \leftarrow A \cdot x, y \leftarrow A^T \cdot t \), a cache-interleaved implementation can be up to \( 1.8 \times \) faster, and up to \( 4.2 \times \) faster than an unblocked two-step implementation [47].

4. **Expose the cost of tuning:** The semantics of the interface make the tuning step and memory costs “logically explicit,” meaning the user decides where and when memory copies and run-time tuning may occur.

   We assume tuning at run-time, the latest point in time at which the matrix will be known completely. However, tuning has a non-trivial execution time cost and must therefore be performed judiciously. Our interface requires the user to call a “tune” routine explicitly, thereby logically exposing the tuning step. This call is optional if the user does not desire tuning, but may also be called repeatedly to re-tune periodically (see Goal 5 below).

   In the case of register blocking, our fully automated heuristic for choosing a near-optimal block size, followed by a conversion from a conventional reference format to the blocked format, can cost as many as 40 SpMVs [49]. Making the tune step explicit helps to avoid surprising overheads in applications that cannot amortize the cost of tuning over many uses of the matrix.

5. **Support self-profiling:** The library may monitor all operations performed on a given matrix and use this information in deciding how aggressively (i.e., how much) to tune. The cost of tuning is acceptable in important application contexts like solving linear systems or computing eigenvalues by iterative methods, where hundreds (or more) of SpMVs can be required for a given matrix [7, 15, 4, 40]. In principle, self-profiling enables the library to guess whether tuning will be profitable (see Section 2.3 on page 12).

6. **Allow for user inspection and control of the tuning process:** Our interface allows a user to examine summaries of which transformations and optimizations were applied during tuning. In addition, the user may (1) save tuning decisions to apply on future problems (*a la* FFTW’s *wisdom* [20]), (2) control the cost of tuning by giving hints about the workload and specify memory usage constraints, and (3) circumvent the automated tuning process by manually specifying what optimizations to apply (see Section 3.5 on page 25).

   We use a library because it enables the use of run-time information for tuning, and because of its potential immediate impact on applications. Our choice of primitives is designed to integrate readily into popular sparse solver libraries such as PETSc [6, 5], or even high-level problem solving environments such as MATLAB [43, 22]. The library-based approach complements other sophisticated techniques for generating sparse kernels, as discussed in Section 6 on page 34.

   ^1Indeed, the ATLAS self-tuning library for the dense BLAS [50] and the FFTW library for discrete Fourier transforms [20] are included in MATLAB.
2 An Introduction to the Tuning Interface by Example

This section introduces the C version of the BeBOP interface by a series of examples. The interface uses an object-oriented calling style, where the two main object types are (1) a sparse matrix object, and (2) a dense (multiple) vector object. This section illustrates three ways in which we envision the interface will be used (Sections 2.1–2.3):

1. **Gradual migration to tunable matrix objects and kernels** (Section 2.1): The interface supports gradual migration for application codes that already use standard array implementations of basic sparse matrix formats (e.g., compressed sparse row (CSR) format and compressed sparse column (CSC) format) and dense vectors. Users do not have to use any of the automatic tuning facilities, or may introduce the use of tuned operations gradually over time.

2. **Tuning using explicit workload and structural hints** (Section 2.2 on page 10): Users can optionally provide the library with information relevant to tuning, such as the expected workload (i.e., which kernels will be used and how frequently), or whether there is special non-zero structure (e.g., uniformly aligned dense blocks, symmetry). The user then calls a special “tune routine” to choose a new data structure performance-optimized for the specified workload.

3. **Tuning using an implicit workload** (Section 2.3 on page 12): Rather than specifying the workload explicitly, a user may rely on the library to monitor kernel calls to determine the workload dynamically. The user still calls the tune routine to perform optimizations, but this routine optimizes based on an inferred workload.

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

Section 3 on page 12 summarizes the interface bindings, and the complete C bindings appear in Appendix B on page 42. We discuss error-handling mechanisms in detail in Section 3.6 on page 29.

2.1 Basic usage: gradual migrating applications

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

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

For demonstration purposes, lines 9–10 declare and initialize two arrays, x and y, to represent the vectors. Again, these declarations are “standard” implementations in that

---

2Fortran interfaces are also available.
Listing 1: A usage example without tuning. This example illustrates basic object creation and kernel execution in our interface. Here, we perform one sparse matrix-vector multiply for a lower triangular matrix $A$ with all ones on the diagonal, as shown in the leading comment.

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

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

// Create a tunable sparse matrix object.
bebop_matrix_t A_tunable = bebop_CreateMat_CSR(
    Aptr, Aind, Aval, 3, 3,
    // CSR arrays
    VIEW_BUFFERS, // "copy mode"
    // remaining args specify how to interpret non-zero pattern
    3, INDEX_ZERO_BASED, MATRIX_TRI_LOWER, MATRIX_UNIT_DIAG IMPLICIT );

// Create wrappers around the dense vectors.
bebop_vecview_t x_view = bebop_CreateVecView( x, 3, VECTOR_UNIT_STRIDE );
bebop_vecview_t y_view = bebop_CreateVecView( y, 3, VECTOR_UNIT_STRIDE );

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

// Clean-up interface objects
bebop_DestroyMat( A_tunable );
bebop_DestroyVecView(x_view);
bebop_DestroyVecView(y_view);

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

2.1 Basic usage: gradual migrating applications
the user could call the dense BLAS on these arrays to perform, for instance, dot products
or scalar-times-vector products ("axpy" operations in the BLAS terminology).

We create a tunable matrix object, $A_{\text{tunable}}$, from the input matrix by a call to \texttt{bebop-CreateMatCSR} (lines 15–19) with the following arguments:

1. Arguments 1–3 specify the CSR arrays (line 16).
2. Arguments 4–5 specify the matrix dimensions (line 16).
3. The 6th argument to \texttt{bebop.CreateMatCSR} (line 17) specifies the copy mode of the object. Here, the value \texttt{VIEW_BUFFERS} tells $A_{\text{tunable}}$ to keep a "shallow copy" (or view) of the CSR arrays $\mathbf{Aptr}$, $\mathbf{Aind}$, and $\mathbf{Aval}$. That is, $A_{\text{tunable}}$ is conceptually a wrapper around these arrays.
4. Arguments 7–10 tell the library how to interpret the CSR arrays (lines 19). Argument 7 is a count that says the next 3 arguments are semantic properties needed to interpret the input matrix correctly. First, \texttt{INDEX_ZERO_BASED} says that the index values in $\mathbf{Aptr}$ and $\mathbf{Aind}$ follow the C convention of starting at 0, as opposed to the typical Fortran convention of starting at 1 (the default is 1-based indexing if not otherwise specified). The value \texttt{MATRIX_TRI_LOWER} asserts the pattern is lower triangular and \texttt{MATRIX_UNIT_DIAG_IMPLICIT} asserts that no diagonal elements are specified explicitly but should be taken to be 1. The library may, at this call, check these properties to ensure they are true if the cost of doing so is $O$(no. of non-zeros).

By specifying \texttt{VIEW_BUFFERS}, the user promises that $\mathbf{Aptr}$, $\mathbf{Aind}$, and $\mathbf{Aval}$ will point to a valid input matrix with a fixed non-zero pattern throughout the lifetime of $A_{\text{tunable}}$, i.e., at least until the call to \texttt{bebop_DestroyMat} (line 29). Other modes are discussed in Section 3.2 on page 14. Since no tuning occurs in this example, the user can be sure $A_{\text{tunable}}$ will not create any copies of the input matrix.

The routine \texttt{bebop.CreateMatCSR} actually accepts a variable number of arguments; only the first 6 arguments are required. If the user does not provide the optional arguments, the library assumes the defaults discussed in Section 3.2 on page 14.

Dense vector objects of type \texttt{bebop.vecview_t}, are always views (shallow copies) of user arrays (lines 22–23). A vector view encapsulates basic information about an array, such as its length, or such as the stride between consecutive elements of the vector within the array. As with the BLAS, a non-unit stride allows a dense vector to be a submatrix. In addition, an object of type \texttt{bebop.vecview_t} can encapsulate multiple vectors—or, a multivector—for kernels like sparse matrix-multiple vector multiply (SpMM) or triangular solve with multiple simultaneous right-hand sides. The multivector object would also store the number of vectors and the memory organization (i.e., row vs. column major), as discussed Section 3.2.3 on page 17. Requiring the user to create a view in both the single- and multiple-vector cases helps unify and simplify some of the kernel argument lists.

The argument lists to kernels, such as \texttt{bebop_MatMult} for SpMV in this example (line 26), follow some of the conventions of the dense BLAS. For example, a user can specify the constant \texttt{OP_TRANS} as the second argument to apply $A^T$ instead of $A$, or specify other values for $\alpha$ and $\beta$.

The calls to \texttt{bebop_DestroyMat} and \texttt{bebop_DestroyVecView} free any memory allocated by the library to these objects (lines 29–31). However, since $A_{\text{tunable}}$, $x_{\text{view}}$, \texttt{bebop_DestroyVecView}
and \( y_{\text{view}} \) are views on user data, the user is responsible for deallocating the associated buffers (\( A_{\text{ptr}}, A_{\text{ind}}, A_{\text{val}}, x, \) and \( y \)).

That \( A_{\text{tunable}}, x_{\text{view}}, \) and \( y_{\text{view}} \) are views implies the user can continue to operate on the data to which these views point as she normally would. For instance, the user can call dense BLAS operations, such as a dot products or scalar-vector multiply (the so-called “axpy” operation), on \( x \) and \( y \), as shown in the biconjugate gradient example of Section 4 on page 30. Moreover, the user might choose to introduce calls to the BeBOP kernels selectively, or gradually over time.

### 2.2 Providing explicit tuning hints

The user tunes a sparse matrix object by optionally providing one or more tuning “hints,” followed by an explicit call to the matrix tuning routine, \texttt{bebop\_TuneMat}. Hints describe the expected workload, or assert performance-relevant structural properties of the matrix non-zeros.

Listing 2 on the next page sketches a simple example in which we provide two tuning hints. The first hint, made via a call to \texttt{bebop\_SetHint\_MatMult}, specifies the expected workload. We refer to such a hint as a \textit{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, \texttt{bebop\_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 \texttt{SYMBOLIC\_VECTOR} indicates that we will apply the matrix to a single vector with unit stride. Alternatively, we could use the constant \texttt{SYMBOLIC\_MULTIVECTOR} to indicate that we will perform SpMM on at least two vectors. Better still, we could pass an actual instance of a \texttt{bebop\_vecview\_t} object which has the precise stride and data layout information. Analogous routines exist for each of the other kernels in the system.

The second hint, made via a call to \texttt{bebop\_SetHint}, is a \textit{structural hint} telling the library that we believe that the matrix non-zero structure is dominated by a single block size. Several of the possible structural hints accept optional arguments that may be used to qualify the hint—for this example, the user might explicitly specify a block size, though here she instead uses the constant \texttt{ARGS\_UNSPECIFIED} to avoid doing so. The library implementation might then know to try register blocking since it would be most likely to yield the fastest implementation [28]. We describe a variety of other hints in Section 3.4 on page 20. These hints are directly related to candidate optimizations explored in our work, and we expect the list of hints to grow over time.

The actual tuning (\textit{i.e.}, possible change in data structure) occurs at the call to \texttt{bebop\_TuneMat}. This example happens to execute SpMV exactly 500 times, though there is certainly no requirement to do so. Indeed, a user may specify one of the symbolic constants \texttt{CALLS\_OFTEN}, or even \texttt{CALLS\_VERY\_OFTEN}, instead of specifying an exact number to tell the library to go ahead and assume the application can amortize cost. (The definition of “often” will depend on the particularly library implementation and hardware platform.)

Once \( A_{\text{tunable}} \) has been created, a user may call the tuning hints as often as and
Listing 2: An example of basic explicit tuning. This example creates a sparse matrix object 
\texttt{A\_tunable} and then tunes it for a workload in which we expect to call SpMV 500 times. In 
addition, we provide an additional hint to the library that the matrix non-zero structure is 
dominated by a dense blocks of a single size, uniformly aligned. Later in the application, 
we actually call SpMV a total of 500 times in some doubly nested loop.

\begin{verbatim}
// Create a tunable sparse matrix object.
A\_tunable = bebop\_CreateMat\_CSR( ... );

// Tell the library we expect to perform 500 SpMV operations with \( \alpha = 1, \beta = 1 \).
bebop\_SetHint\_MatMult( A\_tunable,
   OP\_NORMAL, 1.0, SYMBOLIC\_VECTOR, 1.0, SYMBOLIC\_VECTOR,
   500 ); // workload hint
bebop\_SetHint( A\_tunable, HINT\_SINGLE\_BLOCKSIZE,
   ARG5\_UNSPECIFIED ); // structural hint
bebop\_TuneMat( A\_tunable );

// ...
{
   bebop\_vecview\_t x\_view = bebop\_CreateVecView( ... );
   bebop\_vecview\_t y\_view = bebop\_CreateVecView( ... );

   for( i = 0; i < 100; i++ ) {
      // ...
      for( k = 0; k < 5; k++ ) {
         // ...
         bebop\_MatMult( A\_tunable, OP\_NO\_TRANS,
            1.0, \_x\_view, 1.0, \_y\_view );
         // ...
      }
   }
}
\end{verbatim}
whenever she chooses. For example, suppose the user mixes calls to SpMV and $A^T A \cdot x$ in roughly equal proportion. The user can specify such a workload as follows:

```c
  bebop_SetHint_MatMult( A_tunable, ..., 1000 );
  bebop_SetHint_MatTransMatMult( A_tunable, ..., 1000 );
  // ... other hints ...
  bebop_TuneMat( A_tunable );
```

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

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

```c
  bebop_SetHint_MatMult( A_tunable, ..., 2000 );
```

is equivalent to the two-call sequence

```c
  bebop_SetHint_MatMult( A_tunable, ..., 1000 );
  bebop_SetHint_MatMult( A_tunable, ..., 1000 );
```

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

### 2.3 Tuning based on implicit profiling

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

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

Hints may be ignored completely by the library, so the precise behavior when specifying hints, particularly if they are interleaved between executions of `bebop_TuneMat`, cannot precisely defined. We provide some guidelines in Section 3.4 on page 20.

### 3 Interface

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

1. Creating and modifying sparse matrix and dense vector objects (Section 3.2 on page 14; Table 1 on page 15): A sparse matrix object must be created from an existing user-allocated, `pre-assembled` matrix. We refer to this user-assembled matrix as the *input matrix*. (Appendix A on page 40 defines currently supported input matrix formats.) The user may specify whether the matrix object is a deep copy or a shallow copy of the input matrix. When the library “tunes” a matrix object, it may choose a new internal representation (sparse data structure).

Dense vector objects are wrappers around user-allocated dense arrays.
Listing 3: An example of implicit tuning. This example calls \texttt{bebop\_TuneMat} periodically, without explicitly providing any hints. At each call to \texttt{bebop\_TuneMat}, the library potentially knows more and more about how the user is using \texttt{A\_tunable} and may therefore tune accordingly.

\begin{verbatim}
bebop\_matrix\_t \texttt{A\_tunable} = \texttt{bebop\_CreateMat\_CSR( ... )};
bebop\_vecview\_t \texttt{x\_view} = \texttt{bebop\_CreateVecView( ... )};
bebop\_vecview\_t \texttt{y\_view} = \texttt{bebop\_CreateVecView( ... )};
bebop\_SetHint( \texttt{A\_tunable, HINT\_SINGLE\_BLOCKSIZE, 6, 6} );

\# ...

for( i = 0; i < num\_times; i++ ) {
    \# ...
    while( !converged ) {
        \# ...
        \texttt{bebop\_MatMult( A\_tunable, OP\_NORMAL, 1.0, x\_view, 1.0, y\_view );}
        \# ...
    }
    \texttt{bebop\_TuneMat( A\_tunable );}
    \# \ldots maybe change a few non-zero values for the next solve \ldots
}
\end{verbatim}

2. Executing kernels (Section 3.3 on page 18; Table 3 on page 19), e.g., sparse matrix-vector multiply, triangular solve: The interfaces to our kernel routines mimic the “look-and-feel” of the BLAS.

3. Tuning (Section 3.4 on page 20; Table 6 on page 21): Tuning occurs only if and when the user calls a particular routine in our interface. In addition to this “tune” routine, we also provide auxiliary routines that allow users to provide optional tuning hints.

4. Saving and restoring tuning transformations (Section 3.5 on page 25; Table 12 on page 25): We provide a routine to allow the user to see a precise description, represented by a string, of the transformations that convert the input matrix data structure to the tuned data structure.

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

The interface itself does not define the format of these string-based transformations. However, we suggest a procedural, high-level scripting language, BeBOP-Lua (derived from the Lua language [26]), for representing such transformations. We provide a high-level overview of BeBOP-Lua in Section 5 on page 32.
5. **Error-handling** (Section 3.6 on page 29; Table 13 on page 30): In addition to the error codes and values returned by every routine in the interface, a user may optionally specify her own handler to be called when errors occur to access additional diagnostic information.

Tables 1–13 summarize the available routines. A user who only needs BLAS-like kernel functionality for her numerical algorithms or applications only needs to know about the object creation and kernel routines (Categories 1 and 2 above). Although tuning (i.e., Categories 3 and 4) is an important part of our overall design, its use is strictly optional.

The C bindings are presented in detail in Appendix B on page 42. The following text provides an overview of the semantics and intent behind these bindings.

### 3.1 Basic scalar types

Most sparse matrix formats require storing both floating-point data for non-zero values and integer index data. Our interface is defined in terms of two scalar types accordingly: `bebop_value_t` and `bebop_index_t`. By default, these types are bound to double and int, respectively, but may be overridden at library build-time.

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

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

### 3.2 Creating and modifying matrix and vector objects

Our interface defines two basic abstract data types for matrices and vectors: `bebop_matrix_t` and `bebop_vecview_t`, respectively. Available primitives to create and manipulate objects of these types appears in Table 1 on the following page, and C bindings appear in Appendix B.1 on page 42.

#### 3.2.1 Creating matrix objects

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

At present, we support 0- and 1-based CSR and CSC representations for the input matrix. For detailed definitions of valid input formats, refer to Appendix A on page 40.

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

```c
A_tunable = bebop_CreateMat_CSR( Aptr, Aind, Aval, 3, 3, <copymode>,
                                  <k>, <property_1>, ..., <property_k>);
```

where `A_tunable` is the newly created matrix object, `Aptr`, `Aind`, and `Aval` are user created arrays that store the input matrix (here in a valid CSR format), `<copymode>` specifies how
3.2 Creating and modifying matrix and vector objects

<table>
<thead>
<tr>
<th>Matrix objects</th>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bebop_CreateMat_CSR</td>
<td>Create a valid, tunable matrix object from a CSR input matrix.</td>
</tr>
<tr>
<td></td>
<td>bebop_CreateMat_CSC</td>
<td>Create a valid, tunable matrix object from a CSC input matrix.</td>
</tr>
<tr>
<td></td>
<td>bebop_CopyMat</td>
<td>Clone a matrix object.</td>
</tr>
<tr>
<td></td>
<td>bebop_DestroyMat</td>
<td>Free a matrix object.</td>
</tr>
<tr>
<td></td>
<td>bebop_GetMatEntry</td>
<td>Get the value of a specific matrix entry.</td>
</tr>
<tr>
<td></td>
<td>bebop_SetMatEntry</td>
<td>Set the value of a specific non-zero entry.</td>
</tr>
<tr>
<td></td>
<td>bebop_GetMatClique</td>
<td>Get a block of values, specified as a clique.</td>
</tr>
<tr>
<td></td>
<td>bebop_SetMatClique</td>
<td>Change a block of non-zero values specified as a clique.</td>
</tr>
<tr>
<td></td>
<td>bebop_GetMatDiagValues</td>
<td>Get values along a diagonal of a matrix.</td>
</tr>
<tr>
<td></td>
<td>bebop_SetMatDiagValues</td>
<td>Change values along a diagonal.</td>
</tr>
<tr>
<td></td>
<td>bebop_UpdateMatFromView</td>
<td>Notify the library to update the values in the tuned data structure for a matrix view.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector objects</th>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bebop_CreateVecView</td>
<td>Create a view object for a single vector.</td>
</tr>
<tr>
<td></td>
<td>bebop_CreateMultiVecView</td>
<td>Create a view object for a multivector.</td>
</tr>
<tr>
<td></td>
<td>bebop_CopyVecView</td>
<td>Clone a vector view object.</td>
</tr>
<tr>
<td></td>
<td>bebop_DestroyVecView</td>
<td>Free a (multi)vector view object.</td>
</tr>
</tbody>
</table>

Table 1: Creating and modifying matrix and vector objects. Bindings appear in Appendix B.1 on page 42.

the library should copy the input matrix data, and \(<property_1>\) through \(<property_k>\) specify how the library should interpret that data.

To help the user control memory usage, we support three data copy modes, in which the user tells the library whether to make shallow or deep copies of this data. The three modes, defined by the scalar type `bebop_copymode_t`, are:

1. **VIEW_BUFFERS**: The library should make a shallow copy (or “view”) of the input matrix arrays. The user promises that these buffers point to a matrix with the same non-zero pattern for the lifetime of the new matrix object. If, upon tuning, the library discovers that the best implementation is to keep the matrix in this format, then the library will not keep any extra copies of the matrix. However, since the library shares these buffers with the user, the user must manage the synchronization of these buffers carefully. If instead tuning leads to a change of data structure, the user should call the BeBOP routines to change non-zero values to keep this tuned data structure up-to-date.

2. **COPY_BUFFERS**: The library makes an explicit copy (i.e., a “deep copy”) of the input matrix, and the user may free the input arrays immediately upon return from this call.

3. **OWN_BUFFERS**: The user transfers ownership of the memory buffers to the new matrix object, and the library may free these buffers by a call to the C `free()` routine at


any time. In particular, if, upon tuning, the library decides to replace the input matrix with a new data structure, it can free these buffers. The user should not reference the input matrix buffers after creating the sparse matrix object.

Properties (<property_1> through <property_k> in this example) are optional, and the user should specify as many as needed for the library to properly interpret the non-zero pattern. For instance, Listing 1 on page 8 creates a matrix with implicit ones on the diagonal which are not stored, so the user must specify MATRIX_UNIT_DIAG IMPLICIT as a property. A list of available properties appears in Table 2 on the following page, where default properties assumed by the library are marked with a red asterisk (*).

The user may create a copy of A_tunable by calling bebop_CopyMat. This copy is logically equivalent to creating a matrix object in the COPY BUFFERS mode. The user frees A_tunable or its copies by a call to bebop_DestroyMat.

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

3.2.2 Changing matrix non-zero values

The non-zero pattern of the input matrix fixes the non-zero pattern of A_tunable, but the user may modify the non-zero values. If the input matrix contains explicit zeros, the library treats these entries as logical non-zeros whose values may be modified later. We provide several routines to change non-zero values. To change individual entries, the user may call bebop_SetMatEntry, and to change a block of values defined by a clique, the user may call bebop_SetMatClique. If A_tunable is a shallow copy of the user’s matrix, the user’s values array will also change. Logical non-zero values are subject to properties asserted at matrix creation-time (see Appendix B.1 on page 42).

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

Tuning may select a new data structure in which explicit zero entries are stored that were implicitly 0 (i.e., not stored) in the input matrix. The behavior if the user tries to change these entries is not defined, for two reasons. First, allowing the user to change these entries would yield inconsistent behavior across platforms for the same matrix, since whether a “filled-in” entry could be changed would depend on what data structure the library chooses. Second, requiring that the library detect all such attempts to change these entries might, in the worst case, require keeping a copy of the original input matrix pattern, creating memory overhead. The specifications in Appendix B.1 on page 42 allow, but do not require, the library implementation to report attempts to change implicit zeros to non-zero values as errors.

If A_tunable is a shallow copy, it is possible that the user might change her underlying array data, thus creating a possible inconsistency between this data and any “tuned” copies. If only the non-zero values have changed, the user may call bebop_UpdateMat-
### 3.2 Creating and modifying matrix and vector objects

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MATRIX</strong></td>
<td></td>
</tr>
<tr>
<td>GENERAL</td>
<td>Input matrix specifies all non-zeros.</td>
</tr>
<tr>
<td>TRI_UPPER</td>
<td>Only non-zeros in the upper triangle exist.</td>
</tr>
<tr>
<td>TRI_LOWER</td>
<td>Only non-zeros in the lower triangle exist.</td>
</tr>
<tr>
<td>SYMM_UPPER</td>
<td>Matrix is symmetric but only the upper triangle is stored.</td>
</tr>
<tr>
<td>SYMM_LOWER</td>
<td>Matrix is symmetric but only the lower triangle is stored.</td>
</tr>
<tr>
<td>SYMM_FULL</td>
<td>Matrix is symmetric and all non-zeros are stored.</td>
</tr>
<tr>
<td>HERM_UPPER</td>
<td>Matrix is Hermitian but only the upper triangle is stored.</td>
</tr>
<tr>
<td>HERM_LOWER</td>
<td>Matrix is Hermitian but only the lower triangle is stored.</td>
</tr>
<tr>
<td>HERM_FULL</td>
<td>Matrix is Hermitian and all non-zeros are stored.</td>
</tr>
<tr>
<td>DIAG_EXPLICIT</td>
<td>Any non-zero diagonal entries are specified explicitly.</td>
</tr>
<tr>
<td>UNIT_DIAG_IMPLICIT</td>
<td>No diagonal entries are stored, but should be assumed to be equal to 1.</td>
</tr>
<tr>
<td><strong>INDEX</strong></td>
<td></td>
</tr>
<tr>
<td>ONE_BASED</td>
<td>Array indices start at 1 (default Fortran convention).</td>
</tr>
<tr>
<td>ZERO_BASED</td>
<td>Array indices start at 0 (default C convention).</td>
</tr>
<tr>
<td>UNSORTED</td>
<td>Non-zero indices in CSR (CSC) format within each row (column) appear in any order.</td>
</tr>
<tr>
<td>SORTED</td>
<td>Non-zero indices in CSR (CSC) format within each row (column) are sorted in increasing order.</td>
</tr>
<tr>
<td>REPEATED</td>
<td>Indices may appear multiple times.</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>Indices are unique.</td>
</tr>
</tbody>
</table>

Table 2: **Input matrix properties.** Upon the call to create a matrix object, the user may characterize the input matrix by specifying one or more of the above properties. Properties grouped within the same box are mutually exclusive. Default properties marked by a red asterisk (*).

**FromView** to indicate that a tuned copy, if it exists, should be updated immediately—in the worst case, the cost of executing this routine will be the cost of re-converting the data structure, and the user should bear this cost in mind. If the arrays holding the non-zero pattern have changed, the behavior of the library on **A.tunable** is not defined, though this condition may generate an error if detected, as discussed in the binding for **bebop_UpdateMatFromView** (Appendix B.1 on page 42). The user should take care to maintain consistency between these data structures.

#### 3.2.3 Vector objects

Vector objects (type **bebop_vecview_t**) are always views on the user’s dense array data. Such objects may be views of either single column vectors, created by a call to **bebop_CreateVecView**, or multiple column vectors (multivectors), created by a call to **bebop-**
3.3 Executing kernels

A multivector consisting of \( k \geq 1 \) vectors of length \( n \) each is just a dense \( n \times k \) matrix, but we use the term multivector to suggest a common case in applications in which \( k \) is on the order of a “small” constant (e.g., 10 or less). A single vector is the same as the multivector with \( k = 1 \).

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

In addition to user-created vector views, we define two special, immutable vector view objects: SYMBOLIC_VECTOR and SYMBOLIC_MULTIVECTOR. Conceptually, these objects are constants that may be used with the tuning workload specification routines to indicate tuning for single vectors or multivectors instead of specifying instantiated view objects. See Section 3.4 on page 20.

3.3 Executing kernels

We summarize the available kernels in Table 3 on the next page, and present their bindings in Appendix B.3 on page 57. In addition to both single vector and multivector versions of sparse matrix-vector multiply (bebop_MatMult) and sparse triangular solve (bebop_MatTrisolve), we provide interfaces for three “high-level” sparse kernels that provide more opportunities to reuse the elements of \( A \):

- Simultaneous multiplication of \( A \) and \( A^T \) (or \( A^H \)) by a dense multivector (bebop_MatMult_and_MatTransMult).
- Multiplication of \( A^T \cdot A \) or \( A \cdot A^T \) (or conjugate transpose variants) by a dense multivector (bebop_MatTransMatMult).
- Multiplication of a non-negative integer power of a matrix (bebop_MatPowMult).

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

3.3.1 Applying the transpose of a matrix

We follow the BLAS convention of allowing the user to apply the transpose (or, for complex data, the transpose or Hermitian transpose). See Table 4 on the next page for a list of transpose options provided by the scalar type bebop_matop_t. The notation \( \text{op}(A) \) indicates that any of \( A, A^T, \) or \( A^H \) may be applied, i.e., \( \text{op}(A) \in \{ A, A^T, A^H \} \).

The high-level kernel bebop_MatTransMatMult has inherently more matrix reuse opportunities. This kernel allows the user to apply any of the four matrix operations listed in Table 5 on the following page, given a matrix \( A \): \( A A^T, A^T A, A A^H, \) and \( A^H A \).

3.3.2 Aliasing

The interface guarantees correct results only if multivector view object input arguments do not alias any multivector view object output arguments, i.e., if input and output views do
### Table 3: Sparse kernels

This table summarizes all of the available sparse kernel routines. The user selects single or multivector versions by passing in an appropriate vector view (Section 3.2.3 on page 17). See Appendix B.3 on page 57 for bindings.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bebop_MatMult</strong></td>
<td>Sparse matrix-vector multiply (SpMV)</td>
</tr>
<tr>
<td></td>
<td>( y \leftarrow \alpha \cdot \text{op}(A) \cdot x )</td>
</tr>
<tr>
<td></td>
<td>where ( \text{op}(A) \in {A, A^T, A^H} ).</td>
</tr>
<tr>
<td><strong>bebop_MatTrisolve</strong></td>
<td>Sparse triangular solve (SpTS)</td>
</tr>
<tr>
<td></td>
<td>( x \leftarrow \alpha \cdot \text{op}(A)^{-1} \cdot x )</td>
</tr>
<tr>
<td><strong>bebop_MatTransMatMult</strong></td>
<td>( y \leftarrow \alpha \cdot \text{op}_2(A) \cdot x + \beta \cdot y )</td>
</tr>
<tr>
<td></td>
<td>where ( \text{op}_2(A) \in {A^T A, A^H A, AA^T, AA^H} ).</td>
</tr>
<tr>
<td><strong>bebop_MatMult_and_MatTransMult</strong></td>
<td>Simultaneous computation of</td>
</tr>
<tr>
<td></td>
<td>( y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y )</td>
</tr>
<tr>
<td></td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>( z \leftarrow \omega \cdot \text{op}(A) \cdot w + \zeta \cdot z )</td>
</tr>
<tr>
<td><strong>bebop_MatPowMult</strong></td>
<td>Matrix power multiplication</td>
</tr>
<tr>
<td></td>
<td>Computes ( y \leftarrow \alpha \cdot \text{op}(A)^\rho \cdot x + \beta \cdot y )</td>
</tr>
</tbody>
</table>

### Table 4: Matrix transpose options (type `bebop_matop_t`)

Constants that allow a user to apply a matrix \( A \), its transpose \( A^T \), or, for complex-valued matrices, its conjugate transpose \( A^H \).

<table>
<thead>
<tr>
<th>OP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP_NORMAL</td>
<td>Apply ( A ).</td>
</tr>
<tr>
<td>OP_TRANS</td>
<td>Apply ( A^T ).</td>
</tr>
<tr>
<td>OP_CONJ_TRANS</td>
<td>Apply ( A^H = A^T ), the conjugate transpose of ( A ).</td>
</tr>
</tbody>
</table>

### Table 5: Matrix-transpose-times-matrix options (type `bebop_ataop_t`)

Constants that allow a user to apply \( A^T A \), \( A^H A \), \( AA^T \), or \( AA^H \) in calls to the routine, `bebop_MatTransMatMult`.

<table>
<thead>
<tr>
<th>OP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP_AT_A</td>
<td>Apply ( A^T A ).</td>
</tr>
<tr>
<td>OP_AH_A</td>
<td>Apply ( A^H A ).</td>
</tr>
<tr>
<td>OP_A_AT</td>
<td>Apply ( AA^T ).</td>
</tr>
<tr>
<td>OP_A_AH</td>
<td>Apply ( AA^H ).</td>
</tr>
</tbody>
</table>
not view the same user data. If such aliasing occurs, the results are not defined.

3.3.3 Scalars vs. 1x1 matrix objects

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

3.3.4 Compatible dimensions for matrix multiplication

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

3.3.5 Floating point exceptions

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

3.4 Tuning

The user tunes a valid matrix object at any time and as frequently as she desires for a given matrix object of type `bebop_matrix_t`. The library tunes by selecting a data structure customized for the user’s matrix, kernel workload, and machine. The interface defines three groups of tuning operations, listed in Table 6 on the next page and summarized as follows:

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

- **Structural hint specification** (Section 3.4.2 on page 23): The user may optionally influence the tuning decisions by providing hints about the non-zero structure of the matrix. For example, the user may tell the library that she believes the structure of the matrix consists predominantly of uniformly aligned $6 \times 6$ dense blocks.

- **Explicit tuning** (Section 3.4.3 on page 24): The user must explicitly call the “tune routine,” `bebop_TuneMat`, to initiate tuning. Conceptually, this routine marks the point in program execution at which the library may spend time changing the data structure. The tune routine uses any hints about the non-zero structure or workload, whether they are specified explicitly by the user via calls to the above tuning primitives or they are gathered implicitly during any kernel calls made during the lifetime of the matrix object.

---

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

Workload hints specify the expected options and frequency of the corresponding kernel call.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bebop_SetHint_MatMult</code></td>
<td>Specify hints about the non-zero structure that may be relevant to tuning.</td>
</tr>
<tr>
<td><code>SetHint_MatTrisolve</code></td>
<td>For a list of available hints, see Table 7 on the next page.</td>
</tr>
<tr>
<td><code>SetHint_MatTransMatMult</code></td>
<td>Tune the matrix data structure using all hints and implicit workload data</td>
</tr>
<tr>
<td><code>SetHint_MatPowMult</code></td>
<td>accumulated so far.</td>
</tr>
<tr>
<td><code>SetHint</code></td>
<td>The user tunes a matrix object by first specifying workload and structural</td>
</tr>
<tr>
<td></td>
<td>hints, followed by an explicit call to the tuning routine, <code>bebop_TuneMat</code>.</td>
</tr>
<tr>
<td></td>
<td>Bindings appear in Appendix B.4 on page 62.</td>
</tr>
<tr>
<td><code>TuneMat</code></td>
<td>Section 2 on page 7 illustrates the common ways in which we expect users to</td>
</tr>
<tr>
<td></td>
<td>use the interface to tune.</td>
</tr>
</tbody>
</table>

The library may optimize kernel performance by permuting the rows and columns of the matrix to reduce the bandwidth [27, 44, 13, 23, 14] or to create dense block structure [36]. That is, the library may compute a tuned matrix representation \( \hat{A} = P_r \cdot A \cdot P_c^T \) for the user’s matrix \( A \), where \( P_r \) and \( P_c \) are permutation matrices. However, this optimization requires each kernel to permute its vectors on entry and exit to maintain the correctness of the interfaces. Section 3.4.4 on page 24 discusses functionality that allows the user, if she so desires, to determine if reordering has taken place and access \( P_r, P_c^T \), and \( \hat{A} \) directly to reduce the number of permutations.

### 3.4.1 Providing workload hints explicitly

Each of the kernels listed in Table 3 on page 19 has a corresponding workload hint routine. The user calls these routines to tell the library which kernels she will call and with what arguments for a given matrix object, and the expected frequency of such calls. The routines for specifying workload hints (Table 6) all have an argument signature of the form

```c
bebop_SetHint_<KERNEL>( A_tunable, <KERNEL_PARAMS>, num_calls );
```

where `num_calls` is an integer. This hint tells the library that we will call the specified \( \langle \text{KERNEL} \rangle \) on the object \( A_{\text{tunable}} \) with the arguments \( \langle \text{KERNEL}\_\text{PARAMS} \rangle \), and that we expect to make \( \text{num\_calls} \) such calls.

Instead of specifying an estimate of the number of calls explicitly, the user may substitute the symbolic constant `CALLS_OFTEN` or `CALLS_VERY_OFTEN` to tell the library to go ahead and assume the application can amortize cost (see Table 8 on page 23. The use of two constants allows a library implementation to provide two levels of tuning when the user cannot estimate the number of calls.

Where a kernel expects a vector view object to be passed as an argument, the user may pass to the workload hint either `SYMBOLIC\_VECTOR` or `SYMBOLIC\_MULTIVEC-`
<table>
<thead>
<tr>
<th>Hint</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HINT_NO_BLOCKS</td>
<td>none</td>
<td>Matrix contains little or no dense block substructure.</td>
</tr>
<tr>
<td>HINT_SINGLE_BLOCKSIZE</td>
<td>[int (r, c)]</td>
<td>Matrix structure is dominated by a single block size, (r \times c).</td>
</tr>
<tr>
<td>HINT_MULTIPLE_BLOCKSIZES</td>
<td>[int (k, r_1, c_1, \ldots, r_k, c_k)]</td>
<td>Matrix structure consists of at least (k \geq 1) multiple block sizes. These sizes include (r_1 \times c_1, \ldots, r_k \times c_k).</td>
</tr>
<tr>
<td>HINT_ALIGNED_BLOCKS</td>
<td>none</td>
<td>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).</td>
</tr>
<tr>
<td>HINT_UNALIGNED_BLOCKS</td>
<td>none</td>
<td>Any dense blocks are not aligned, or the alignment is unknown.</td>
</tr>
<tr>
<td>HINT_SYMM_PATTERN</td>
<td>none</td>
<td>The matrix non-zero pattern is structurally symmetric, or nearly so.</td>
</tr>
<tr>
<td>HINT_NONSYMM_PATTERN</td>
<td>none</td>
<td>The matrix non-zero pattern is structurally “very” unsymmetric.</td>
</tr>
<tr>
<td>HINT_RANDOM_PATTERN</td>
<td>none</td>
<td>The matrix non-zeros (or non-zero blocks) are nearly distributed uniformly randomly over all positions.</td>
</tr>
<tr>
<td>HINT_CORRELATED_PATTERN</td>
<td>none</td>
<td>The row indices and column indices for non-zeros are highly correlated.</td>
</tr>
<tr>
<td>HINT_NO_DIAGS</td>
<td>none</td>
<td>The matrix contains little if any explicit diagonal structure.</td>
</tr>
<tr>
<td>HINT_DIAGS</td>
<td>[int (k, d_1, \ldots, d_k)]</td>
<td>The matrix has structure best represented by multiple diagonals. The diagonal lengths include (d_1, \ldots, d_k), possibly among other lengths.</td>
</tr>
</tbody>
</table>

Table 7: Available structural hints (type `bebop_tunehint_t`). The user may provide additional hints, via a call to the routine `bebop_SetHint`, about the non-zero structure of the matrix which may be useful to tuning. Several of the hints allow the user to specify additional arguments, shown in column 2. All arguments are optional. The table groups hints into 5 mutually exclusive sets, e.g., a user should only specify one of HINT_NO_BLOCKS, HINT_SINGLE_BLOCKSIZE, and HINT_MULTIPLE_BLOCKSIZES if she specifies any of these hints at all.
3.4 Tuning

<table>
<thead>
<tr>
<th>CALLS_OFTEN</th>
<th>The user expects “many” calls, and the library may therefore elect to do some basic tuning.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALLS_VERY_OFTEN</td>
<td>The user expects a sufficient number of calls that the library may tune aggressively.</td>
</tr>
</tbody>
</table>

Table 8: **Symbolic calling frequency constants (type int)**. Instead of providing a numerical estimate of the number of calls the user expects to make when specifying a workload hint, the user may use one of the above symbolic constants.

<table>
<thead>
<tr>
<th>SYMBOLIC_VECTOR</th>
<th>A symbolic single vector view.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYMBOLIC_MULTIVECTOR</td>
<td>A symbolic multivector view consisting of at least two vectors.</td>
</tr>
</tbody>
</table>

Table 9: **Symbolic vector views for workload hints (type bebop_vecview_t)**. Instead of passing an actual vector view object to the workload hint routine (Table 6 on page 21), the user may pass in one of the above symbolic views.

TOR instead of an actual vector view object (Table 9). The user should use SYMBOLIC_VECTOR if she anticipates using a single vector, or SYMBOLIC_MULTIVECTOR if she anticipates using at least two vectors. Specifying actual vector view objects is preferred since they will contain additional information relevant to tuning, including storage layout for multivectors (i.e., row vs. column major) and strides or leading dimensions.

3.4.2 Providing structural hints

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

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

```
bebop_SetHint( A_tunable, HINT_SINGLE_BLOCKSIZE, ARGS_UNSPECIFIED );
```

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

```
bebop_SetHint( A_tunable, HINT_SINGLE_BLOCKSIZE, 6, 6 ); // 6 × 6 blocks
```

In this case, either call is “correct” since specifying the block size is optional. See Table 7 on the preceding page for a list of hints, their arguments, and whether the arguments are optional or required.

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

- If more than one hint from a mutually exclusive group is specified, assume the latter is true. For example, if the user specifies HINT_SINGLE_BLOCKSIZE followed by HINT_NO_BLOCKS, then no-block hint should override the single-block size hint.
TUNESTAT_NEW | The library selected a new data structure for
the matrix based on the current workload
data and hints.

TUNESTAT_AS_IS | The library did not change the data structure.

Table 10: Tuning status codes. Status codes returned by `bebop.TuneMat` in the event that
no error occurred during tuning.

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

3.4.3 Initiating tuning

This interface defines a single routine, `bebop.TuneMat`, which marks the point in program execution at which tuning may occur. As discussed in its binding (Section B.4 on page 62), `bebop.TuneMat` returns one of the integer status codes shown in Table 10 to indicate whether it changed the data structure (TUNESTAT_NEW) or not (TUNESTAT_AS_IS).

3.4.4 Accessing the permuted form

The interface defines several routines (Table 11 on the next page) that allow the user to determine whether the library has optimized kernel performance by reordering the rows and columns of the matrix (by calling `bebop.IsMatPermuted`), to extract the corresponding permutations (`bebop.ViewPermutedMat, bebop_ViewPermutedMat_RowPerm, bebop_ViewPermutedMat_ColPerm`), and to apply these permutations to vector view objects (`bebop_PermuteVecView`).

Given the user’s matrix $A$, suppose that tuning produces the representation $A = P^T_r \cdot \hat{A} \cdot P_c$, where $P_r$ and $P_c$ are permutation matrices and multiplying by $\hat{A}$ is much faster than multiplying by $A$. This form corresponds to reordering the rows and columns of $A$—by pre- and post-multiplying $A$ by $P_r$ and $P^T_c$—to produce an optimized matrix $\hat{A}$. To compute $y \leftarrow A \cdot x$ correctly while maintaining its interface and taking advantage of fast multiplication by $\hat{A}$, the kernel `bebop_MatMult` must instead compute $P^T_r \cdot y \leftarrow \hat{A} \cdot (P_c \cdot x)$. Carrying out this computation in-place requires permuting $x$ and $y$ on entry, and then again on return. If tuning produces such a permuted matrix, all kernels perform these permutations as necessary.

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

We provide an additional type for permutations, `bebop.perm_t`, and the routines listed in Table 11 on the following page. An object of type `bebop.perm_t` may equal a special
3.5 Saving and restoring tuning transformations

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bebop_IsMatPermuted</td>
<td>Determine whether a matrix has been tuned by reordering its rows or columns.</td>
</tr>
<tr>
<td>bebop_ViewPermutedMat</td>
<td>Returns a read-only matrix object for the re-ordered copy of $\hat{A}$, $\hat{A}$.</td>
</tr>
<tr>
<td>bebop_ViewPermutedMat_RowPerm</td>
<td>Returns the row permutation $P_r$.</td>
</tr>
<tr>
<td>bebop_ViewPermutedMat_ColPerm</td>
<td>Returns the column permutation $P_c$.</td>
</tr>
<tr>
<td>bebop_PermuteVecView</td>
<td>Apply a permutation object (or its inverse/transpose) to a vector view.</td>
</tr>
</tbody>
</table>

Table 11: **Extracting and applying permuted forms.** If tuning produces a tuned matrix $\hat{A} = P_r \cdot A \cdot P_c^T$, the above routines allow the user to detect and to extract read-only views of $P_r$, $P_c$, and $\hat{A}$, and apply the permutations $P_r$ and $P_c$. Bindings appear in Appendix B.5 on page 67.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bebop_GetMatTransforms</td>
<td>Retrieve a string representation of the tuning transformations that have been applied to a given matrix.</td>
</tr>
<tr>
<td>bebop_ApplyMatTransforms</td>
<td>Apply tuning transformations to a given matrix.</td>
</tr>
</tbody>
</table>

Table 12: **Saving and restoring tuning transformations.** The interface defines a basic facility to allow users to view the tuning transformations that have been applied to matrix, and later re-apply those (or other) transformations to another matrix. Bindings appear in Appendix B.6 on page 69.

symbolic constant representing an identity permutation of any size, **PERM_IDENTITY**. This constant may be used in either of the routines to apply a permutation or its inverse to a vector view. Listing 4 on the next page sketches the way in which a user might use these routines in her application.

3.5 Saving and restoring tuning transformations

The interface defines basic facilities that allow users to view the tuning transformations which have been applied to a matrix, to edit these transformations, and to re-apply them (Table 12). To get a string describing how the input matrix data structure was transformed during tuning, the user calls **bebop_GetMatTransforms**. This routine returns a newly allocated string containing the transformations description. To set the data structure (i.e., to apply some set of transformations to the input matrix data structure), the user calls **bebop_ApplyMatTransforms**. Such a call is equivalent to calling **bebop_TuneMat**, except that instead of allowing the library to decide what data structure to use, we are specifying it explicitly. We illustrate the usage of these two routines in Listing 5 on page 27 and Listing 6 on page 28.

We do not mandate the precise format of the string, but strongly encourage the use of a human-readable format. Section 5 on page 32 provides several examples of transformations expressed in a high-level procedural language based on Lua [26].
Listing 4: An example of extracting permutations.

// Computes $y \leftarrow A^k \cdot x$, where $A = \begin{pmatrix} 0.25 & 0 & 0 \\ 0 & 0.75 & 0 \\ 0.75 & 0.25 & 1 \end{pmatrix}$.
int Aptr[] = { 1, 2, 3, 6 }, Aind[] = { 1, 2, 1, 2, 3 };  // 1-based, CSR format.
double Aval[] = { 0.25, 0.75, 0.75, 0.25, 1.0 };  // 1-based, CSR format.
int iter, max_power = 3;  // k

// Create vectors $x^T = (1 \ 1 \ 1)$ and $y = 0$.
double x[] = { 1, 1, 1 }, y[] = { 0, 0, 0 };

// Store permuted form. Declared as ‘const’ since they will be read-only.
const bebop_perm_t Pr, Pc;  // Stores $P_r, P_c$
const bebop_matrix_t A_fast;  // Stores $A$

// Tune for our computation
bebop_SetHint_MatMult( A_tunable, OP_NORMAL, 1, x_view, 1, y_view, max_power );
bebop_TuneMat( A_tunable );

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

// Sanity check: if the matrix was not reordered for performance, then
// $P_r = P_c = I$, the identity matrix.
if ( bebop_IsMatPermuted( A_tunable ) ) {
    assert ( Pr == PERM.IDENTITY ); assert( Pc == PERM.IDENTITY );
    assert( A_fast == A_tunable );
}

// Compute $y \leftarrow A^k \cdot x$ in three steps.
// 1. $y \leftarrow P_r \cdot y, x \leftarrow P_c \cdot x$
bebop_PermuteVecView( Pr, OP_NORMAL, y_view );
bebop_PermuteVecView( Pc, OP_NORMAL, x_view );

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

// 3. $y \leftarrow P_r^T \cdot y, x \leftarrow P_c^T \cdot x$
bebop_PermuteVecView( Pr, OP_TRANS, y_view );
bebop_PermuteVecView( Pc, OP_TRANS, x_view );

// Clean-up
bebop_DestroyMat( A_tunable );  // Invalidates Pr, Pc, and A_fast
bebop_DestroyVecView( x_view );
bebop_DestroyVecView( y_view );

// Should print, "y = A^3*x = [ 0.015625 ; 0.421875 ; 2.5625 ];"
printf ( "y = A^%d*x = [ %f ; %f ; %f ];\n", max_power, y[0], y[1], y[2] );
Listing 5: An example of saving transformations. This example extracts the tuning transformations applied to a matrix object \texttt{A\_tunable} and saves them to a file.

```c
bebop\_matrix\_t \texttt{A\_tunable} =bebop\_CreateMat\_CSR(\ldots);

char* xforms; // stores transformations
FILE* fp\_saved\_xforms = fopen("./my\_xform\.txt", "wt"); // text file for output

// ...

// Tune the matrix object
bebop\_TuneMat( \texttt{A\_tunable} );

// ...

// Extract transformations
xforms = bebop\_GetMatTransforms( \texttt{A\_tunable} );
printf( "--- Matrix transformations ---\n%s\n--- end ---\n", xforms );

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

free( xforms );

// ...
```

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

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

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

bebop_matrix_t A_tunable = bebop_CreateMat_CSR( ... );

num_chars_read = fread( xforms, sizeof(char), SOME_MAX_BUFFER_SIZE-1, fp_saved_xforms );
xforms[num_chars_read] = NULL;

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

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

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

// ...
```
3.6 Handling errors

The BeBOP interface provides two methods for handling errors:

1. **Error code returns**: Many of the routines in the interface return an integer error code (of type int). All of the possible error codes are negative, providing a simple way for an application to check for an error on return from any BeBOP routine.

2. **Error handling routines**: The user may associate every matrix object with its own error handling routine. When a matrix object $A_{\text{tunable}}$ is created, it is automatically associated with the library’s default error handling routine, bebop_HandleError_Default. However, the user may change the handler at any time, and even set it to NULL to disable error handling when it occurs with $A_{\text{tunable}}$.

In addition, the library maintains a global error handler, which is initialized to be bebop_HandleError_Default. The library calls the global error handler if $A_{\text{tunable}}$ is unavailable or invalid. The user may change global error handler at any time, or set it to NULL to disable error handling entirely.

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

- If the user passed a valid matrix object $A_{\text{tunable}}$ to the routine, the routine calls $A_{\text{tunable}}$’s error handler. Recall that this handler is initially bebop_HandleError_Default. If $A_{\text{tunable}}$ is valid but its handler has been set to NULL, no error handling occurs.

- If $A_{\text{tunable}}$ is unavailable or invalid, the routine calls the current global error handler.

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

A user may change the error handler by calling bebop_SetErrorHandler, or retrieve the current matrix-specific or global error handler by calling bebop_GetErrorHandler. The error handling routines are summarized in Table 13 on the following page.

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

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

The first 5 parameters describe the error and its source location. The arguments beginning at format_string are printf-compatible arguments that provide a flexible way to provide supplemental error information.

For example, the following code shows how the error handler might be called from within the the SpMV kernel, bebop_MatMult, when the user incorrectly tries to apply a matrix $A_{\text{tunable}}$ with dimensions $m \times n$ to a vector of length veclen, where $n \neq \text{veclen}$:

```c
if ( n != veclen ) {
    your_handler( A_{\text{tunable}}, ERR_MIN_DIM_MISMATCH,
                  "bebop_MatMult: Mismatched dimensions", "bebop_MatMult.c", 507,
```
### Table 13: Error handling routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bebop_GetErrorHandler</code></td>
<td>Returns a pointer to the current error handling routine for a given matrix object, or to the current global error handler if the caller does not supply a matrix.</td>
</tr>
<tr>
<td><code>bebop_SetErrorHandler</code></td>
<td>Changes the current error handling routine for a given matrix object, or changes the current global error handler if the caller does not supply a matrix.</td>
</tr>
<tr>
<td><code>bebop_HandleError_Default</code></td>
<td>The default error handler provided by the library.</td>
</tr>
</tbody>
</table>

**Table 13: Error handling routines.** Bindings appear in Appendix B.7 on page 70.

```c
"Matrix dimensions: %d x %d, Vector length: %d\n", m, n, veclen );
return ERR猱DIM.IMISMATCH;
```

## 4 Example: Biconjugate Gradients

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

```c
// Solves the \( n \times n \) system \( A \cdot x = b \) for \( x \) using the BiCG algorithm.
// The vector \( x \) should be initialized with a starting guess.
int SolveLinSys_using_BiCG( const bebop_matrix_t A, int n, const double* b, double* x )
{
    static const int STRIDE_1 = 1;

    int converged = 0; // == 1 when algorithm has converged.
    double *workspace; // Temporary vector storage space.
    double *p, *ps, *y, *ys, *r, *rs; // Temporary vectors of length \( n \) each

    // For BeBOP
    // Argument error checking
    assert( n > 0 );
    assert( A != INVALID_MATRIX );
    assert( x != NULL );
    assert( b != NULL );

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

    p = workspace; ps = workspace + n;
    y = workspace + 2*n; ys = workspace + 3*n;
    r = workspace + 4*n; rs = workspace + 5*n;
```
Example: Biconjugate Gradients

```c
p_view = bebob_CreateVecView( p, n, STRIDE_1 );
ps_view = bebob_CreateVecView( ps, n, STRIDE_1 );
y_view = bebob_CreateVecView( y, n, STRIDE_1 );
ys_view = bebob_CreateVecView( ys, n, STRIDE_1 );
r_view = bebob_CreateVecView( r, n, STRIDE_1 );
rs_view = bebob_CreateVecView( rs, n, STRIDE_1 );

// Compute residual r_0 ← b, r_0 ← r_0 - A·x_0
dcopy( n, b, STRIDE_1, r, STRIDE_1 );
bebop_MatMult( A, OP_NORMAL, -1.0, x_view, 1.0, r_view );
dcopy( n, r, STRIDE_1, rs, STRIDE_1 );
dcopy( n, r, STRIDE_1, p, STRIDE_1 );
dcopy( n, r, STRIDE_1, ps, STRIDE_1 );
rs_view = bebob_CreateVecView( rs, n, STRIDE_1 );

while( !converged ) {
    // Inner loop, iteration j (starting at j = 0)
    double r_dot_rs, alpha, beta;
    dcopy( n, r, STRIDE_1, rs, STRIDE_1 ); // r_j^* ← r
    dcopy( n, r, STRIDE_1, p, STRIDE_1 ); // p_j^* ← r
    dcopy( n, r, STRIDE_1, ps, STRIDE_1 ); // p_j^* ← r

    // Simultaneously compute: y ← A·p_j, y^* ← A^T·p_j^*
    bebob_MatMult_and_MatTransMult( A, 1.0, p_view, 0.0, y_view, OP_TRANS, 1.0, ps_view, 0.0, ys_view );
    r_dot_rs = ddot( n, r, STRIDE_1, rs, STRIDE_1 );
    alpha = r_dot_rs / ddot(n, y, STRIDE_1, ps, STRIDE_1);
    daxpy( n, alpha, p, STRIDE_1, x, STRIDE_1 ); // x_{j+1} ← x_j + α_j · p_j
    daxpy( n, alpha, y, STRIDE_1, r, STRIDE_1 ); // r_{j+1} ← r_j - α_j · A·p_j
    daxpy( n, -alpha, ys, STRIDE_1, r, STRIDE_1 ); // r_{j+1}^* ← r_j^* - α_j · A^T·p_j^*
    beta = ddot(n, r, STRIDE_1, rs, STRIDE_1) / r_dot_rs; // β_j ← (r_j^T · r_{j+1}^*) / (r_j^T · r_j^*)
    dscal( n, beta, p, STRIDE_1 );
    daxpy( n, 1.0, r, STRIDE_1, p, STRIDE_1 );
    dscal( n, beta, ps, STRIDE_1 );
    daxpy( n, 1.0, rs, STRIDE_1, ps, STRIDE_1 );

    // Check for convergence
    converged = ...
}

bebop_DestroyVecView( r_view );
bebop_DestroyVecView( x_view );
bebop_DestroyVecView( y_view );
bebop_DestroyVecView( ys_view );
bebop_DestroyVecView( p_view );
bebop_DestroyVecView( ps_view );
free( workspace );
```
This example does not explicitly tune, but if this solver is embedded as a subroutine call in some larger non-linear solver, one could imagine calling \texttt{bebop\_TuneMat} just after a call to this routine.

5 A Tuning Transformation Language

This section provides an overview, illustrated by example, of a dynamically typed, procedural object-oriented scripting language for describing how to convert an input matrix into a tuned matrix data structure. The syntax of the language, BeBOP-Lua, is based on the scripting language Lua \cite{26}. A call to \texttt{bebop\_GetMatTransforms} (Section 3.5 on page 25) returns a program in this language. Moreover, a user may write her own sequence of transformations as a program in this language, and then call \texttt{bebop\_ApplyMatTransforms} to execute the program, thereby creating a specific data structure.

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

5.1 Basic transformations

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

```lua
define No tuning
    return InputMat;
end
```

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

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

```lua
define Convert input matrix to $4 \times 2$ register blocked format
    return BCSR.new( InputMat, 4, 2 );
end
```

Here, \texttt{BCSR} is a type and \texttt{new} is a method that takes a matrix object and block size as input, and returns a matrix object in the new format.

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

5.2 A complex splitting example

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

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

```plaintext
# Let \( A = A_1 + A_2 + A_3 \) where
# \( A_1 \) is in \( 4 \times 2 \) UBCSR format
# \( A_2 \) is in \( 2 \times 2 \) UBCSR format
# \( A_3 \) is in CSR format
T = VBR.new( InputMat );

# First, split \( A = A_1 + A_{\text{leftover}} \),
# where \( A_{\text{leftover}} \) is in CSR format
A1, A_{\text{leftover}} = T.extract_blocks( 4, 2 );

# Next, split \( A_{\text{leftover}} = A_2 + A_3 \)
T = VBR.new( A_{\text{leftover} } );
A2, A3 = T.extract_blocks( 2, 2 );

return A1 + A2 + A3;
```

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

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

The \texttt{VBR} type has a method, \texttt{extract_blocks}, which greedily extracts blocks of the spec-
fied size, and returns two matrix objects: one in UBCSR containing all blocks, and the
other in CSR to hold the leftover elements.\(^4\) This method is first called (line 9) to extract
\( 4 \times 2 \) blocks, and repeated to extract \( 2 \times 2 \) blocks from the leftovers (line 13).

The summation in the return statement (line 15) is a \textit{symbolic} summation. The expres-
sion \( A_1 + A_2 + A_3 \) implicitly evaluates to an object of type \texttt{SUM}. Indeed, this statement is
equivalent to creating an object via

```
return SUM.new( A1, A2, A3 );
```

A garbage collector automatically disposes of the temporary variables, \texttt{T} and \texttt{A_{\text{leftover}}}.

\(^4\) A naturally occurring \( 5 \times 4 \) block will be extracted as two \( 4 \times 2 \) blocks, with the leftover elements assigned
to \texttt{A_{\text{leftover}}}.
The function `reorder_TSP` (line 2) executes our implementation of Pinar and Heath’s TSP-based reordering algorithm [36]. The first output variable, `A_hat`, stores the reordered matrix in CSR format. The second argument holds the row permutation. Since a column permutation is not assigned on output, `reorder_TSP` will apply the same permutation to both the rows and the columns.

The call to `extract_blocks` (line 5) uses a greedy algorithm to select $2 \times 2$ blocks from `A_hat`, and returns the result in BCSR format in `A1`. This `extract_blocks` method, a member of the CSR type, differs from the method of the same name used in the example of Section 5.2 on page 32, since that method was a member of the VBR (and not CSR) type.

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

```plaintext
A_tuned = PERMFORM:new( transpose(P), SUM:new(A1, A2), P );
```

5.4 Switch-to-dense example

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

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

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

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

```plaintext
1 ASSERT( is_lower(InputMat) );
2 L1, L2, LD = split_s2d( InputMat );
3 return TRIPART:new( L1, L2, LD );
```

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

```plaintext
1 ASSERT( is_lower(InputMat) );
2 switch_point = 25382;  \# $L_2$ and $L_D$ begin at row 25382
3 L1, L2, LD = split_s2d( InputMat, switch_point );
4 return TRIPART:new( L1, L2, LD );
```

6 Approaches that Complement Libraries

There are a number of complementary approaches to a library implementation. One is to implement a library using a language with generic programming constructs such as templates in C++ [34]. This approach has been adopted Blitz++ [45] and the Matrix Template
Library (MTL) [41] to build generic libraries in C++ that mimic dense BLAS functionality. The use of templates facilitates the generation of large numbers of library routines with relatively small amount of code, and flexibly handles issues of producing libraries that can handle different precisions. Sophisticated use of templates furthermore allows some limited optimization, such as unrolling. In some cases, loop-fusion like transformations have been implemented using templates [45]. However, this approach lacks an explicit mechanism for dealing with run-time search. Furthermore, the template mechanism for code generation can put enormous stress (in terms of memory and execution time) on the compiler.\(^5\)

Another approach which extends the generic programming idea is compiler-based sparse code generation via restructuring compilers, pursued by Bik [8, 9, 10], Stodghill, et al. [42, 2, 32, 31], and Pugh and Shpeisman [37, 29]. These are clean, general approaches to code generation: the user expresses separately both the kernels (as dense code with random access to matrix elements) and a formal specification of a desired sparse data structure; a restructuring compiler combines the two descriptions to produce a sparse implementation. In addition, since any kernel can in principle be expressed, this overcomes a library approach in which all possible kernels must be pre-defined. Nevertheless, we view this technology as complementary to the overall library approach: while sparse compilers could be used to provide the underlying implementations of sparse primitives, they do not explicitly make use of matrix structural information available, in general, only at run-time.\(^6\)

A third approach is to extend an existing library or system. There are a number of application-level libraries (e.g., PETSc [6, 5], among others [21, 39, 38, 24]) and high-level application tools (e.g., MATLAB [43, 22], Octave [18], approaches that apply compiler analyses and transformations to MATLAB code [3, 33]) that provide high-level sparse kernel support. Integration with these systems has a number of advantages, including the ability to hide data structure details and the tuning process from the user, and the large potential user base. However, our goal is to provide building blocks in the spirit of the BLAS with the steps and costs of tuning exposed. This model of development has been very successful with other numerical libraries, examples of which include the integration of ATLAS and FFTW tuning systems into the commercial MATLAB system. Thus, it should be possible to integrate the BeBOP library into an existing system as well.

\(^5\)This concern is “practical” in nature and could be overcome through better compiler front-end technology. Another minor but related concern is the lack of consistency in how well aspects of the template mechanism are supported, making portability an issue.

\(^6\)Technically, Bik’s sparse compiler does use matrix non-zero structure information [10], but is restricted in the following two senses: (1) it assumes that the matrix is available at “compile-time,” and (2) it supports a limited number of fixed data structures.
References


A Valid input matrix representations

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

- **Packed 3-array compressed sparse row using 0-based indices**: The user provides 3 arrays, $A_{ptr}$, $A_{ind}$, and $A_{val}$ corresponding to $A$. These arrays satisfy the following conditions:

  1. $A_{ptr}$ is of length at least $A_{ptr}[m+1]$, $A_{ptr}[0] \geq 0$, and for all $0 \leq i < m$, $A_{ptr}[i] \leq A_{ptr}[i+1]$.
  2. $A_{ind}$ is of length at least $A_{ptr}[m]$. Each element of $A_{ind}$ lies between 0 and $n-1$ inclusive.
  3. $A_{val}$ is of length at least $A_{ptr}[m]$.

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

- **Packed 3-array compressed sparse row using 1-based indices**: The user provides 3 arrays, $A_{ptr}$, $A_{ind}$, and $A_{val}$ corresponding to $A$. These arrays satisfy the following conditions:

  1. $A_{ptr}$ is of length at least $A_{ptr}[m+1]$, $A_{ptr}[0] \geq 1$, and for all $0 \leq i < m$, $A_{ptr}[i] \leq A_{ptr}[i+1]$.
  2. $A_{ind}$ is of length at least $A_{ptr}[m]$. Each element of $A_{ind}$ lies between 1 and $n$ inclusive.
  3. $A_{val}$ is of length at least $A_{ptr}[m]$.

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

- **Packed 3-array compressed sparse column using 0-based indices**: The user provides 3 arrays, $A_{ptr}$, $A_{ind}$, and $A_{val}$ corresponding to $A$. These arrays satisfy the following conditions:

  1. $A_{ptr}$ is of length at least $A_{ptr}[n+1]$, $A_{ptr}[0] \geq 0$, and for all $0 \leq j < n$, $A_{ptr}[j] \leq A_{ptr}[j+1]$.
  2. $A_{ind}$ is of length at least $A_{ptr}[n]$. Each element of $A_{ind}$ lies between 0 and $m-1$ inclusive.
  3. $A_{val}$ is of length at least $A_{ptr}[n]$.

  A matrix element $A(i, j)$ is computed from this representation as follows. Let $K_{ij}$ be the set $\{ k : A_{ptr}[j-1] \leq k < A_{ptr}[j] \text{ and } A_{ind}[k] = i-1 \}$. Then $A(i, j) = \sum_{k \in K} A_{val}[k]$. (Repeated elements are summed.)
• **Packed 3-array compressed sparse column using 1-based indices**: The user provides 3 arrays, Aptr, Aind, and Aval corresponding to $A$. These arrays satisfy the following conditions:

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

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

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

```c
int factorial ( int n );
```

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

**Parameters:**

- \( n \) [input]  
  
  Non-negative integer of which to compute a factorial.

**Actions and Returns:**

An integer whose value equals \( n! \) if \( n \) is greater than 1, or 1 if \( n \) equals 0. The return value is undefined if \( n! \) exceeds the maximum positive integer of type int.

**Error conditions and actions:**

Aborts program if \( n \) is less than 0.

**Example:**

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

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

As discussed in Section 3.6 on page 29, the interface provides two error-handling mechanisms: return codes and error-handling functions. By convention, readers can assume that any routine returning integers (type int) will return negative values on errors. In addition, all routines call an error handler if one is available in a given context according to the process described in Section 3.6. For all routines, a violation of argument preconditions is always considered an error condition.

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

### B.1 Matrix object creation and modification

```c
bebop_matrix_t
bebop_CreateMat_CSR(
    bebop_index_t* Aptr, bebop_index_t* Aind, bebop_value_t* Aval,
    bebop_index_t num_rows, bebop_index_t num_cols,
    bebop_copymode_t mode,
)```
int k, [bebop_inmatprop_t property_1, ..., bebop_inmatprop_t property_k]);

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

**Parameters:**

- **num_rows × num_cols** [input]  
  Dimensions of the input matrix.

- **Aptr, Aind, Aval** [input]  
  The input matrix pattern and values must correspond to a valid CSR representation, as defined in Appendix A on page 40.

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

- **k** [input]  
  The number of qualifying properties.

- **property_1, ..., property_k** [input; optional]  
  See Table 2 on page 17. The user may assert that the input matrix satisfies zero or more properties listed in Table 2 on page 17. Grouped properties are mutually exclusive, and specifying two or more properties from the same group generates an error (see below). The user must supply exactly \( k \) properties.

**Actions and Returns:**

- A valid, tunable matrix object, or **INVALID_MATRIX** on error. Any kernel operations or tuning operations may be called using this object.

**Error conditions and actions:**

Possible error conditions include:

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

**Example:**  
See Listing 1 on page 8.

```c
bebop_matrix_t
bebop_CreateMat_CSC(
    bebop_index_t* Aptr, bebop_index_t* Aind, bebop_value_t* Aval,
    bebop_index_t num_rows, bebop_index_t num_cols,
    bebop_copymode_t mode,
    [bebop_inmatprop_t property_1, ..., bebop_inmatprop_t property_k]);
```

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

**Parameters:**
\textbf{num\_rows} × \textbf{num\_cols} [input] \quad \textbf{num\_rows} ≥ 0, \textbf{num\_cols} ≥ 0

Dimensions of the input matrix.

\textbf{Aptr}, \textbf{Aind}, \textbf{Aval} [input] \quad \textbf{Aptr}, \textbf{Aind}, \textbf{Aval} \neq \text{NULL}

The input matrix pattern and values must correspond to a valid CSC representation, as defined in Appendix A on page 40.

\textbf{mode} [input]

Specifies the copy mode for the arrays \textbf{Aptr}, \textbf{Aind}, and \textbf{Aval}. See Section 3.2.1 on page 14 for a detailed explanation.

\textbf{property\_1, \ldots property\_k} [input; optional] \quad \text{See Table 2 on page 17.}

The user may assert that the input matrix satisfies zero or more properties listed in Table 2 on page 17. Grouped properties are mutually exclusive, and specifying two or more properties from the same group generates an error (see below).

\textbf{Actions and Returns:}

A valid, tunable matrix object, or \texttt{INVALID\_MATRIX} on error. Any kernel operations or tuning operations may be called using this object.

\textbf{Error conditions and actions:}

Possible error conditions include:

1. Any of the argument preconditions above are not satisfied [\texttt{ERR\_BAD\_ARG}].
2. More than 1 property from the same group are specified (see Table 2 on page 17) [\texttt{ERR\_IN-MATPROP\_CONFLICT}].
3. The input matrix arrays do not correspond to a valid CSC representation [\texttt{ERR\_NOT\_CSC}], or are incompatible with any of the asserted properties [\texttt{ERR\_FALSE\_INMATPROP}].

\begin{verbatim}
bebop\_value\_t
bebop\_GetMatEntry( const bebop\_matrix\_t A\_tunable, bebop\_index\_t row, bebop\_index\_t col );
\end{verbatim}

Returns the value of a matrix element.

\textbf{Parameters:}
\begin{itemize}
  \item \textbf{A\_tunable} [input] \quad \texttt{A\_tunable} is valid.
  \item \textbf{row, col} [input] \quad \texttt{1 ≤ row ≤ m, 1 ≤ col ≤ n}
\end{itemize}

The object representing some \( m \times n \) matrix \( A \).

 specifies the element whose value is to be returned. The precondition above must be satisfied. Note that matrix entries are referenced using 1-based indices, regardless of the convention used when the matrix was created.

\textbf{Actions and Returns:}

If \texttt{row} and \texttt{col} are valid, then this routine returns the value of the element \( A(\texttt{row, col}) \). Otherwise, it returns \texttt{NaN\_VALUE}.

\textbf{Error conditions and actions:}

Possible error conditions include:

1. Invalid matrix [\texttt{ERR\_BAD\_MATRIX}].
2. Position \texttt{row, col} is out-of-range [\texttt{ERR\_BAD\_ENTRY}].
Example:

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

bebop_value_t
bebop_SetMatEntry( bebop_matrix_t A_tunable, bebop_index_t row, bebop_index_t col,
                   bebop_value_t val );

Changes the value of the specified matrix element.

Parameters:

A_tunable [input/output] A_tunable is valid
The object representing some \( m \times n \) matrix \( A \).

row, col [input] \( 1 \leq \text{row} \leq m, 1 \leq \text{col} \leq n \)
Specifies the element whose value is to be modified. This element must have had an associated element stored explicitly in the input matrix when \( A \) was created.

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

Actions and Returns:
Returns 0 and sets \( A(\text{row}, \text{col}) \leftarrow \text{val} \). If the matrix was created as either symmetric or Hermitian (including the semantic properties, MATRIX_SYMM_FULL and MATRIX_HERM_FULL), this routine logically sets \( A(\text{col}, \text{row}) \) to be \( \text{val} \) also. On error, \( A \) remains unchanged and an error code is returned.

NOTE: When \( A \) 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 \( A \) was created.

Error conditions and actions:
Possible error conditions include:

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

Example:

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

bebop_matrix_t A_tunable = bebop_CreateMat_CSR( Aptr, Aind, Aval, 3, 3, VIEW_BUFFERS, 2, MATRIX_SYMM_UPPER, MATRIX_UNIT_DIAG_IMPLICIT);

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

// Change A(3,1) and A(1,3) to -.5.
bebop_SetMatEntry( A_tunable, 3, 1, -0.5);

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

int bebop_GetMatClique( const bebop_matrix_t A_tunable, const bebop_index_t* rows, bebop_index_t num_rows, const bebop_index_t* cols, bebop_index_t num_cols, bebop_vecview_t vals);

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

Parameters:

A_tunable [input]  
A_tunable is valid

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

rows, cols [input]  
rows ≠ NULL, cols ≠ NULL

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

1 ≤ rows[i] ≤ m for all 0 ≤ i < num_rows, and
1 ≤ cols[j] ≤ n for all 0 ≤ j < num_cols.

vals [output]  
vals is valid.

The object vals is a multivector view (see Section 3.2.3 on page 17) of a logical two-dimensional array to be used to store the block of values. We use a view here to allow the user to specify row or column major storage and the leading dimension of the array.

Actions and Returns:

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

Error conditions and actions:

Possible errors conditions include:

1. Invalid matrix [ERR_BAD_MATRIX].
2. An invalid row, col was given [ERR_BAD_ENTRY].
Example:

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

bebop_vecview_t vals_view = bebop_CreateMultiVecView( vals, 2, 2, STORAGE_ROW_MAJOR, 2 );

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

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

```c
int
bebop_SetMatClique( bebop_matrix_t A_tunable,
const bebop_index_t* rows, bebop_index_t num_rows,
const bebop_index_t* cols, bebop_index_t num_cols,
const bebop_vecview_t vals );
```

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

**Parameters:**

- **A_tunable [output]**
  
  The object representing some \( m \times n \) matrix \( A \).

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

- **rows, cols [input]**
  
  Indices defining the block of values. The array \( \text{rows} \) must be of length at least \( \text{num_rows} \), and \( \text{cols} \) must be of length at least \( \text{num_cols} \). The entries of \( \text{rows} \) and \( \text{cols} \) must satisfy
  
  \[
  1 \leq \text{rows}[i] \leq m \quad \text{for all } 0 \leq i < \text{num_rows}, \quad \text{and}
  \]
  
  \[
  1 \leq \text{cols}[j] \leq n \quad \text{for all } 0 \leq j < \text{num_cols}.
  \]

- **vals [input]**
  
  The object \( \text{vals} \) is a multivector view (see Section 3.2.3 on page 17) of a logical two-dimensional array to be used to store the block of values. We use a view here to allow the user to specify row or column major storage and the leading dimension of the array.

**Actions and Returns:**

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

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

If an entry \( A(i,j) \) is specified by the clique and appears multiple times within the clique with inconsistent values in \( \text{vals} \), the behavior is undefined.
NOTE: When A_tunable is tuned, the tuned data structure may store additional explicit zeros to improve performance. The user should avoid changing entries that were not explicitly stored when A_tunable was created.

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

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

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

bebop_matrix_t A_tunable = bebop_CreateMat_CSR( Aptr, Aind, Aval, 3, 3, VIEW_BUFFERS, 2, MATRIX_SYMM_UPPER, MATRIX_UNIT_DIAG.IMPLICIT );

// Clique of values to change, using 1-based indices to match matrix.
// The new values are \begin{pmatrix} 1 & .125 \\ .125 & 1 \end{pmatrix}.
int rows[] = {1, 2};
int cols[] = {1, 2};
double vals[] = {-1, -1, -1, -1}; // in row major order
double new_vals[] = {1, .125, .125, 1 }; // in row major order

bebop_vecview_t vals_view = bebop_CreateMultiVecView( vals, 2, 2, STORAGE_ROW_MAJOR, 2 );
bebop_vecview_t new_vals_view = bebop_CreateMultiVecView( new_vals, 2, 2, STORAGE_ROW_MAJOR, 2 );

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

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

int
bebop_GetMatDiagValues( const bebop_matrix_t A_tunable, bebop_index_t diag_num,
                        bebop_vecview_t diag_vals );

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

Parameters:
\textbf{A\_tunable} [input]
\( A \) is valid.
The \( m \times n \) matrix \( A \) from which to extract diagonal entries.

\textbf{diag\_num} [input]
1\( -m \leq \text{diag\_num} \leq n - 1 \)
Number \( d \) of the diagonal to extract.

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

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

Error conditions and actions:
Possible error conditions include:
1. Providing an invalid matrix \([\text{ERR\_BAD\_MATRIX}]\).
2. Providing an invalid vector view, or a vector view with invalid dimensions \([\text{ERR\_BAD\_-\_VECVIEW}]\).
3. Specifying an invalid diagonal \([\text{ERR\_BAD\_ARG}]\).

Example:
// First, create \( A = \begin{pmatrix} 1 & -2 & 0.5 \\ -2 & 1 & 0 \\ 0.5 & 0 & 1 \end{pmatrix} \), a sparse symmetric matrix with a unit diagonal.

int Aptr[] = \{1, 3, 3, 3\}; \quad \text{// Uses 1-based indices!}
double Aind[] = \{1, 2\};
double Aval[] = \{-2, 0.5\};
double diag_vals[] = \{0, 0, 0\};
bebop_vecview_t diag_vals_view = bebop_CreateVecView( diag_vals, 3, UNIT\_STRIDE );

bebop_matrix_t A\_tunable = bebop_CreateMat\_CSR( Aptr, Aind, Aval, 3, 3, VIEW\_BUFFERS,
                      2, MATRIX\_SYMM\_UPPER, MATRIX\_UNIT\_DIAG\_IMPLICIT );

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

// Prints "First superdiagonal = [-2, 0]"
bebop_GetMatDiagValues( A\_tunable, 1, diag_vals_view);
printf( "First superdiagonal = [%f, %f]\n", diag_vals[0], diag_vals[1] );
B.1 Matrix object creation and modification

// Prints “Second subdiagonal = [0.5]”
bebop_GetMatDiagValues( A_tunable, −2, diag vals view );
printf( "Second subdiagonal = [%f] \n", diag vals[0]);

int
bebop_SetMatDiagValues( bebop_matrix_t A_tunable, bebop_index_t diag num,
const bebop_vecview_t diag vals);

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

Parameters:
A_tunable [input/output] $A_tunable$ is valid.
The $m \times n$ matrix $A$ in which to change diagonal entries.
diag_num [input] $1 − m \leq diag_num \leq n − 1$
Number $d$ of the diagonal to change.
diag vals [output] $diag vals$ is a valid view.
Let $X$ be the $r \times s$ (multi)vector object into which to store the diagonal values, such that $s \geq 1$ and $r$ is at least the length of the diagonal, i.e., $r \geq \min\{\max\{m, n\} − d, \min\{m, n\}\}$.

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

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

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

Error conditions and actions:
Possible error conditions include:
1. Providing an invalid matrix [ERR_BAD_MATRIX].
2. Providing an invalid vector view, or a vector view with invalid dimensions [ERR_BAD_VECVIEW].
3. Specifying an invalid diagonal [ERR_BAD_ENTRY].
4. Specifying the main diagonal when $A_tunable$ was created with MATRIX_UNIT_DIAG_IMPLICIT.

Example:
// First, create $A = \begin{pmatrix} 1 & −2 & 0.5 \\
−2 & 1 & 0.25 \\
0.5 & 0 & 1 \end{pmatrix}$, a sparse symmetric matrix with a unit diagonal.
int Aptr[] = {1, 3, 4, 4}; Aind[] = {1, 2, 3}; // Uses 1-based indices!
double Aval[] = {−2, 0.5, 0.25};
double diag vals[] = { 0, 0, 0 };
bebop_vecview_t diag vals view = bebop_CreateVecView( diag vals, 3, UNIT_STRIDE );
bebop_matrix_t A_tunable = bebop_CreateMat_CSR( Aptr, Aind, Aval, 3, 3, VIEW_BUFFERS,
2, MATRIX_SYMM_UPPER, MATRIX_UNIT_DIAG_IMPLICIT);

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

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

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

int bebop_UpdateMatFromView( bebop_matrix_t A_tunable );

For a matrix that is a view on user data, this routine forces the library to update the values in its
tuned copy (if any) from the view.

Parameters:
A_tunable[input/output] A_tunable is a valid view.
Matrix object to update.

Actions and Returns:
If A_tunable was created using the view (shallow copy) mode, VIEWS_BUFFERS, then this routine
updates its tuned copy if any. If A_tunable is not a view, then no changes occur. In the worst
case, the cost of executing this routine is the same as the cost of reconverting the data structure.
This routine assumes that any changes to the underlying buffers were only made to the non-zero
values, and the behavior of operations on A_tunable is undefined if the underlying non-zero pattern
changed. This routine returns an error code on error.

Error conditions and actions:
Possible error conditions include:
1. Providing an invalid matrix object [ERR_BAD_MATRIX].
2. Calling this routine on a matrix object which is not a view [ERR_NOT_MATVIEW].
3. Violating one of the semantic properties originally specified at create-time [ERR_INMAT-PROP_CONFLICT].
4. A change in the non-zero pattern was detected [ERR_MAT_PATTERN_CHANGED].

Example:
// First, create \( A = \begin{pmatrix} 1 & -2 & .5 \\ -2 & 1 & 0 \\ .5 & 0 & 1 \end{pmatrix} \), a sparse symmetric matrix.
int Aptr[] = {1, 4, 5, 6}, Aind[] = {1, 2, 3, 2, 3}; // Uses 1-based indices
double Aval[] = {1, -2, 0.5, 1, 1}; // Uses 1-based indices

bebop_matrix_t A_tunable = bebop_CreateMat_CSR( Aptr, Aind, Aval, 3, 3, VIEW_BUFFERS,
2, MATRIX_SYMM_UPPER, MATRIX_UNIT_DIAG_IMPLICIT);

// The following should print:
// "A(1,2) == -2"
// "A(2,1) == -2"
printf( "A(1,2) == %f\n", bebop_GetMatEntry(A_tunable, 1, 2) );
printf( "A(2,1) == %f\n", bebop_GetMatEntry(A_tunable, 2, 1) );

// ...
// Suppose we tune, and tuning chooses a new data structure.
bebop_TuneMat( A_tunable );

// Change A(1, 2) to 2.
Aval[1] = 2;

// This change may not be reflected in the new data structure, depending
// on whether the library extracts values from the viewed buffers or
// from the tuned data structure. So, the following might print
// "A(1,2) == -2"
// "A(2,1) == -2"
// or
// "A(1,2) == 2"
// "A(2,1) == 2"
printf( "A(1,2) == %f\n", bebop_GetMatEntry(A_tunable, 1, 2) );
printf( "A(2,1) == %f\n", bebop_GetMatEntry(A_tunable, 2, 1) );

// Signaling a change forces synchronization, provided the viewed
// buffers are not otherwise simultaneously changed by another
// thread.
bebop_UpdateMatFromView( A_tunable );

// So, the following should print:
// "A(1,2) == 2"
// "A(2,1) == 2"
printf( "A(1,2) == %f\n", bebop_GetMatEntry(A_tunable, 1, 2) );
printf( "A(2,1) == %f\n", bebop_GetMatEntry(A_tunable, 2, 1) );

bebop_matrix_t
bebop_CopyMat( const bebop_matrix_t A_tunable );

Creates a copy of a matrix object.

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

Actions and Returns:
Returns a new matrix object, or INVALID_MATRIX on error. The new matrix object is equivalent to the matrix object the user would obtain if she performed the following steps:

1. Re-execute the original call to bebop_CreateMat_CSR/bebop_CreateMat_CSC to create a new, untuned matrix object, A_copy, but make a deep copy, i.e., the copy mode COPY_BUFFERS.
Thus, \texttt{A\_copy} may exist independently of \texttt{A\_tunable} and of any data upon which \texttt{A\_tunable} might depend.

2. Get the tuning transformations that have been applied to \texttt{A\_tunable} by the time of this call. Equivalently, execute \texttt{bebop\_GetMatTransformations(A\_tunable)} and store the result.

3. Apply these transformations to \texttt{A\_copy}.

**Error conditions and actions:**
Possible error conditions include an invalid source matrix object [\texttt{ERR\_BAD\_MATRIX}] or an out-of-memory condition while creating the clone [\texttt{ERR\_OUT\_OF\_MEMORY}].

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

bebop_vecview_t vals_view = bebop_CreateMultiVecView(vals, 2, 2, STORAGE\_ROW\_MAJOR, 2);

bebop_matrix_t A_copy;

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

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

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

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

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

\texttt{bebop\_DestroyMat( bebop\_matrix_t A\_tunable );}

Frees object memory associated with a given matrix object. The object is no longer usable.
B.2 Vector object creation

Parameters:

A_tunable [input/output]  
A_tunable is valid

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

Actions and Returns:

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

Error conditions and actions:

Regardless of the return value, A_tunable should not be used after this call. Also, if A_tunable was created in the OWN_BUFFERS copy mode, then any pointers to the original input matrix buffers should not be used. Possible error conditions include an invalid matrix object [ERR_BAD_MATRIX].

Example:

See Listing 1 on page 8.

B.2 Vector object creation

bebop_vecview_t  
bebop_CreateVecView( bebop_value_t* x, bebop_index_t length,  
bebop_index_t inc );

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

Parameters:

length [input]  
Number of vector elements.  

length $\geq 0$

inc [input]  
Stride, or distance in the user’s dense array, between logically consecutive elements of $x$.  

inc $> 0$

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

Actions and Returns:

Returns a valid vector view object for $x$, or INVALID_VECTOR on error.

Error conditions and actions:

An error occurs if any of the argument preconditions are not satisfied [ERR_BAD_ARG].

Example:

See Listing 1 on page 8.

bebop_vecview_t  
bebop_CreateMultiVecView( bebop_value_t* X,  
bebop_index_t length, bebop_index_t num_vecs,  
bebop_storage_t orient, bebop_index_t lead_dim );
B.2 Vector object creation

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

**Parameters:**

- **length** [input]
  Number of elements in each column vector.

- **num_vecs** [input]
  The number of column vectors, i.e., \( k \) as shown above.

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

- **lead_dim** [input]
  This parameter specifies the leading dimension, as specified in the BLAS standard. The leading dimension is the distance in \( X \) between the first element of each row vector, and must be at least \( \text{num_vecs} \), if **orient** == STORAGE_ROW_MAJOR. If instead **orient** == STORAGE_COL_MAJOR, then the leading dimension is the distance in \( X \) between the first element of each column vector, and must be at least \( \text{length} \).

- **X** [input]
  Pointer to the user’s dense array representation of \( X \). For each \( 1 \leq i \leq \text{length} \) and \( 1 \leq j \leq \text{num_vecs} \), element \( x_{ij} \) (the \( i \)th element of the \( j \)th column, is stored at one of the following positions:
  1. If **orient** == STORAGE_ROW_MAJOR, then \( x_{ij} \) is stored at element \( X[i*\text{lead_dim} + j] \).
  2. If **orient** == STORAGE_COL_MAJOR, then \( x_{ij} \) is stored at element \( X[i + j*\text{lead_dim}] \).

**Actions and Returns:**

Returns a valid multivector view on the data stored in \( X \), or INVALID VECTOR on error.

**Error conditions and actions:**

Returns INVALID VECTOR and calls the global error handler on an error. Possible error conditions include:

1. Any of the above argument preconditions are not satisfied [ERR_BAD_ARG].
2. The leading dimension is invalid for the specified storage orientation [ERR_BAD_LEAD_DIM].

**Example:**

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

double Y[] = { 1, -1, 1, -1, 1, -1 }; // in row-major order
bebop_vecview_t Y_view = bebop_CreateMultiVecView(Y, 2, STORAGE_ROW_MAJOR, 2);

double X[] = { .25, .45, .65, -.25, -.45, -.65 }; // in column-major order
bebop_vecview_t X_view = bebop_CreateMultiVecView(X, 3, STORAGE_COL_MAJOR, 3);

bebop_MatMult(A_tunable, OP_NO_TRANS, 1, X_view, 1, Y_view);
```
// Views no longer needed.
bebop_DestroyVecView( X_view );
bebop_DestroyVecView( Y_view );

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

int
bebop_DestroyVecView( bebop_vecview_t x_view );

Destroy a vector view.

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

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

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

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

bebop_vecview_t
bebop_CopyVecView( const bebop_vecview_t x_view );

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

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

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

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

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

// Make a copy of the original view on x
bebop_vecview_t x_copy_view = bebop_CopyVecView( x_view );

// Dispose of original view
bebop_DestroyVecView( x_view );

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

// Finished with all objects
bebop_DestroyMat( A_tunable );
bebop_DestroyVecView( x_copy_view );
bebop_DestroyVecView( y_view );

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

B.3 Kernels

int
bebop_MatMult( const bebop_matrix_t A_tunable, bebop_matop_t opA,
    bebop_value_t alpha, const bebop_vecview_t x_view,
    bebop_value_t beta, bebop_vecview_t y_view );

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

Parameters:

**A_tunable** [input]  
An object for a matrix $A$.  

**opA** [input]  
Specifies $\text{op}(A)$.  

**alpha, beta** [input]  
Scalar constants $\alpha, \beta$, respectively.  

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

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

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

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

**Example:**
See Listing 1 on page 8.

```c
int bebop_MatTrisolve( const bebop_matrix_t T_tunable, bebop_matop_t opT,
                       bebop_value_t alpha, bebop_vecview_t x_view );
```

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

**Parameters:**
- `T_tunable` [input] $T_tunable$ is valid, square, and triangular. Matrix object for an $n \times n$ upper or lower triangular matrix $T$.
- `opT` [input] See Table 4 on page 19. Specifies $\text{op}(T)$.
- `alpha` [input] Scalar constant $\alpha$.
- `x_view` [input/output] $x\_\text{view}$ is valid. View object for a (multi)vector $x$.

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

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

**Example:**
```c
// Let A_tunable be object corresponding to the the sparse lower triangular matrix A
// shown in Listing 1 on page 8. The following example solves $A \cdot x = b$, where
// $b^T = (1.0 0.35)$

double x[] = { 1.0, 0, 0.35 };
bebop_vecview_t x_view = bebop_CreateVecView(x, 3, 1);
bebop_MatTrisolve( A_tunable, OP_NORMAL, 1.0, x_view );

// Should print the solution, “x == [ 0.1 ; 0.2 ; 0.3 ]”
printf( ”x == [ %f ; %f ; %f ]\n”, x[0], x[1], x[2] );
```
int
bebop_MatTransMatMult( const bebop_matrix_t A_tunable, bebop_ataoop_t opA,
                         bebop_value_t alpha, const bebop_vecview_t x_view,
                         bebop_value_t beta, bebop_vecview_t y_view, bebop_vecview_t t_view );

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

**Parameters:**

- **A_tunable** [input] 
  An object for a matrix \( A \).

- **opA** [input] 
  See Table 5 on page 19.
  Specifies \( \text{op}(A) \).

- **alpha, beta** [input] 
  The scalar constants \( \alpha, \beta \), respectively.

- **x_view** [input] 
  View object for a (multi)vector \( x \).
  
- **y_view** [input/output] 
  View object for a (multi)vector \( y \).

- **t_view** [output] 
  An optional view object for a (multi)vector \( t \).
  \( t \text{\ view} \) may be valid or INVALID\_MATRIX.

**Actions and Returns:**

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

**Error conditions and actions:**

Possible error conditions include unsatisfied argument preconditions [ERR\_BAD\_ARG, ERR\_BAD\_MATRIX, ERR\_BAD\_VECVIEW] and incompatible operand dimensions [ERR\_DIM\_MISMATCH].

**Example:**

```
// Let A_tunable be an object corresponding to the sparse lower triangular matrix
// shown in Figure 1 on page 8, and let \( x^T = (0.1 \ 0.2 \ 0.3) \). The following code computes
// \( t \leftarrow A \cdot x \), and \( y \leftarrow A^T A \cdot x \).

// Set-up vectors
double x[] = { 0.1, 0.2, 0.3 };
bebop_vecview_t x_view = bebop_CreateVecView( x, 3, 1 );

double t[] = { -1, -1, -1 };  
bebop_vecview_t t_view = bebop_CreateVecView( t, 3, 1 );

double y[] = { 1, 1, 1 };  
bebop_vecview_t y_view = bebop_CreateVecView( y, 3, 1 );
```
// Execute kernel: \( t \leftarrow A \cdot x, \ y \leftarrow A^T A \cdot x \)
\( \text{bebop\_MatTransMatMult}( \text{A\_tunable}, \text{OP\_AT\_A}, 1, \text{x\_view}, 0, \text{y\_view}, \text{t\_view}); \)

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

int
\( \text{bebop\_MatMult\_and\_MatTransMult}( \text{const bebop\_matrix\_t A\_tunable}, \) 
\( \text{bebop\_value\_t alpha, const bebop\_vecview\_t x\_view,} \) 
\( \text{bebop\_value\_t beta, bebop\_vecview\_t y\_view,} \) 
\( \text{bebop\_matop\_t opA,} \) 
\( \text{bebop\_value\_t omega, const bebop\_vecview\_t w\_view,} \) 
\( \text{bebop\_value\_t zeta, bebop\_vecview\_t z\_view}; \)

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

Parameters:

\( \text{A\_tunable} [\text{input}] \) \hspace{1cm} \text{A\_tunable} \text{ is valid.}

An object for a matrix \( A \).

\( \text{alpha, beta, omega, zeta} [\text{input}] \) \hspace{1cm} \text{The scalar constants } \alpha, \beta, \omega, \zeta, \text{ respectively.}

\( \text{opA} [\text{input}] \) \hspace{1cm} \text{See Table 4 on page 19.}

Specifies \( \text{op}(A) \).

\( \text{x\_view, w\_view} [\text{input}] \) \hspace{1cm} \text{x\_view, w\_view} \text{ are valid.}

View objects for (multi)vectors \( x \) and \( w \), respectively.

\( \text{y\_view, z\_view} [\text{input/output}] \) \hspace{1cm} \text{y\_view, z\_view} \text{ are valid.}

View objects for (multi)vectors \( y \) and \( z \), respectively.

Actions and Returns:

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

Error conditions and actions:

Possible error conditions include unsatisfied argument preconditions [ERR\_BAD\_ARG, ERR\_BAD\_MATRIX, ERR\_BAD\_VECVIEW] and incompatible operand dimensions [ERR\_DIM\_MISMATCH].

Example:

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

double x[] = { .1, .2, .3 };
bebop_vecview_t x_view = bebop_CreateVecView(x, 3, 1);

double y[] = {−1, −1, −1};
bebop_vecview_t y_view = bebop_CreateVecView(y, 3, 1);

double z[] = {1, 1, 1};
bebop_vecview_t z_view = bebop_CreateVecView(z, 3, 1);

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

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

int bebop_MatPowMult( const bebop_matrix_t A_tunable, bebop_matop_t opA, int power,
   bebop_value_t alpha, const bebop_vecview_t x_view,
   bebop_value_t beta, bebop_vecview_t y_view, bebop_vecview_t T_view );

Computes a power of a matrix times a vector, or y ← α · op(A) · x + β · y. Also optionally computes
T = (t_1 · · · t_{ρ−1}), where t_k ← op(A)^k · x for all 1 ≤ k < ρ.

A_tunable [input]  A_tunable is valid.
An object for a matrix A. If ρ > 1, then A must be square.

opA [input]  See Table 4 on page 19.
Specifies op(A).

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

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

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

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

T_view [output]  T_view is a valid multivector view of at least ρ − 1 vectors, or NULL.
If non-NULL, T_view is a view object for the multivector T = (t_1 · · · t_{ρ−1}).

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

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

**Example:**

```
// First create a matrix $A = \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.75 & 0 \\ 0.75 & 0.25 & 1 \end{bmatrix}$ in CSR format using 1-based indices.
int Aptr[] = { 1, 2, 3, 6 };    // 1-based
int Aind[] = { 1, 2, 1, 2, 3 }; // 1-based
double Aval[] = { 0.25, 0.75, 0.75, 0.25, 1.0};
bebob_matrix_t A_tunable = bebop_CreateMat_CSR( Aptr, Aind, Aval, 3, 3, VIEW_BUFFERS, 0);

// Create a vector $x^T = (1 \ 1 \ 1)$.  
double x[] = { 1, 1, 1 };
bebob_vecview_t x_view = bebop_CreateVecView( x, 3, 1 );

// Result vector $y$  
double y[] = { -1, -1, -1 };  
bebob_vecview_t y_view = bebop_CreateVecView( y, 3, 1 );

// Storage space to keep intermediate vectors, $T = (t_1 \ t_2)$.  
// Initially, let $t_1^T = (0.1 \ 0.1 \ 0.1)$ and $t_2^T = (0.2 \ 0.2 \ 0.2)$.  
double T[] = { 0.1, 0.1, 0.1, 0.2, 0.2, 0.2 }; // in column-major order
bebob_vecview_t T_view = bebop_CreateMultiVecView( T, 3, 2, STORAGE_COL_MAJOR, 3 );

// Compute $y \leftarrow A^3 \cdot x$ and the intermediate vectors $t_1, t_2$.  
bebop_MatPowMult( A_tunable, OP_NORMAL, 3, 1.0, x_view, 0.0, y_view, T_view );

// Print results:  
printf( "t1 = A*x = [ %f ; %f ; %f ];\n", T[0], T[1], T[2] );  
printf( "t2 = A^2*x = [ %f ; %f ; %f ];\n", T[3], T[4], T[5] );  
printf( "y = A^3*x = [ %f ; %f ; %f ];\n", y[0], y[1], y[2] );
```

**B.4 Tuning**

```
int bebob_SetHint_MatMult( bebob_matrix_t A_tunable, bebob_matop_t opA, 
        bebob_value_t alpha, const bebob_vecview_t x_view, 
        bebob_value_t beta, const bebob_vecview_t y_view, 
        int num_calls );
```

Workload hint for the kernel operation `bebop_MatMult` which computes $y \leftarrow \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$, where $\text{op}(A) \in \{ A, A^T, A^H \}$.
B.4 Tuning Parameters:

**A.tunable** [input/output]  
An object for a matrix $A$.

**opA** [input]  
Specifies $\text{op}(A)$.

**alpha, beta** [input]  
Scalar constants $\alpha, \beta$, respectively.

**x.view, y.view** [input]  
Vectors are valid or symbolic (see Table 9 on page 23). View object for a (multi)vector $x$ and $y$, respectively.

**num_calls** [input]  
Non-negative or symbolic (see Table 8 on page 23).  
The number of times this kernel will be called with these arguments.

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

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

Example:  
See Listing 2 on page 11.

```c
int bebop_SetHint_MatTrisolve( bebop_matrix_t T_tunable, bebop_matop_t opT,  
    bebop_value_t alpha, const bebop_vecview_t x_view,  
    int num_calls );
```

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

**Parameters:**

**T.tunable** [input/output]  
Matrix object for an $n \times n$ upper or lower triangular matrix $T$.

**opT** [input]  
Specifies $\text{op}(T)$.

**alpha** [input]  
Scalar constant $\alpha$.

**x.view** [input]  
Vectors are valid or symbolic (see Table 9 on page 23). View object for a (multi)vector $x$.

**num_calls** [input]  
Non-negative or symbolic (see Table 8 on page 23).  
The number of times this kernel will be called with these arguments.
**Actions and Returns:**
Registers the workload hint with \texttt{A.tunable} and returns 0 only if the dimensions of \( \text{op}(T) \) and \( x \) have compatible dimensions. Otherwise, returns an error code.

**Error conditions and actions:**
Possible error conditions include unsatisfied argument preconditions \([\text{ERR\_BAD\_ARG, ERR\_BAD\_MATRIX, ERR\_BAD\_VECVIEW}]\) and incompatible operand dimensions \([\text{ERR\_DIM\_MISMATCH}]\).

```c
int bebop_SetHint_MatTransMatMult( bebop_matrix_t A_tunable, bebop_atomp_t opA,
    bebop_value_t alpha, const bebop_vecview_t x_view,
    bebop_value_t beta, const bebop_vecview_t y_view,
    [const bebop_vecview_t t_view,]
    int num_calls);
```

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

**Parameters:**
\texttt{A.tunable} [input/output] \texttt{A.tunable} is valid.
An object for a matrix \( A \).

\texttt{opA} [input]
Specifies \( \text{op}(A) \).

\texttt{alpha, beta} [input]
The scalar constants \( \alpha, \beta \), respectively.

\texttt{x.view, y.view} [input] \texttt{x.view, y.view} are valid or symbolic (see Table 9 on page 23).
View objects for (multi)vector objects \( x, y \), respectively, for a (multi)vector \( x \).

\texttt{t.view} [input; optional] If specified, may be valid, symbolic, or \texttt{INVALID\_MATRIX}.
An optional view object for a (multi)vector \( t \).

\texttt{num.calls} [input] \texttt{num.calls} is non-negative or symbolic (see Table 8 on page 23).
The number of times this kernel will be called with these arguments.

**Actions and Returns:**
Registers the workload hint with \texttt{A.tunable} and returns 0 only if the argument dimensions are compatible. Otherwise, returns an error code.

**Error conditions and actions:**
Possible error conditions include unsatisfied argument preconditions \([\text{ERR\_BAD\_ARG, ERR\_BAD\_MATRIX, ERR\_BAD\_VECVIEW}]\) and incompatible operand dimensions \([\text{ERR\_DIM\_MISMATCH}]\).

```c
int bebop_SetHint_MatMult_and_MatTransMult( bebop_matrix_t A_tunable,
    bebop_value_t alpha, const bebop_vecview_t x_view,
```
Workload hint for the kernel operation `bebop_MatMult_and_MatTransMult` which computes \( y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y \) and \( z \leftarrow \omega \cdot \text{op}(A) \cdot x + \zeta \cdot z \), where \( \text{op}(A) \in \{ A, A^T, A^H \} \).

Parameters:

- `A_tunable [input/output]`: An object for a matrix \( A \).
- `alpha, beta, omega, zeta [input]`: The scalar constants \( \alpha, \beta, \omega, \zeta \), respectively.
- `opA [input]`: Specifies \( \text{op}(A) \).
- `x_view, y_view, w_view, z_view [input]`: Vectors are valid or symbolic (see Table 9 on page 23). View objects for (multi)vectors \( x, y, w, \) and \( z \), respectively.
- `num_calls [input]`: The number of times this kernel will be called with these arguments.

Actions and Returns:

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

Error conditions and actions:

Possible error conditions include unsatisfied argument preconditions [ERR_BAD_ARG, ERR_BAD_MATRIX, ERR_BAD_VECVIEW] and incompatible operand dimensions [ERR_DIM_MISMATCH].

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

- `A_tunable [input/output]`: An object for a matrix \( A \).
- `opA [input]`: Specifies \( \text{op}(A) \).

See Table 4 on page 19.
power [input]  
Power $\rho$ of the matrix $A$ to apply.

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

$x\_view, y\_view$ [input]  
Vectors are valid or symbolic (see Table 9 on page 23) single vectors. View objects for the vectors $x, y$.

$T\_view$ [input; optional]  
A valid or symbolic multivector, or INVALID\_MATRIX. If specified and not equal to INVALID\_MATRIX, $T\_view$ is either a view object for the multivector $T = (t_1 \cdots t_{\rho-1})$, or SYMBOLIC\_MULTIVECTOR.

$num\_calls$ [input]  
$num\_calls$ is non-negative or symbolic (see Table 8 on page 23). The number of times this kernel will be called with these arguments.

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

**Error conditions and actions:**
Possible error conditions include unsatisfied argument preconditions [ERR\_BAD\_ARG, ERR\_BAD\_MATRIX, ERR\_BAD\_VECVIEW] and incompatible operand dimensions [ERR\_DIM\_MISMATCH].

```c
int bebop\_SetHint( bebop\_matrix\_t A\_tunable, bebop\_tunehint\_t hint[, ...] );
```

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

**Parameters:**

$A\_tunable$ [input/output]  
Matrix object for which to register a structural hint.

$hint$ [input]  
See Table 7 on page 22. User-specifie structural hint. This hint may be followed by optional arguments, as listed and typed in Table 7 on page 22.

**Actions and Returns:**
Returns 0 if the hint is recognized and $A\_tunable$ is valid, or an error code otherwise.

**Error conditions and actions:**
Possible error conditions include an invalid matrix object [ERR\_BAD\_MATRIX], or specifying a hint with the wrong number of hint arguments [ERR\_BAD\_HINT\_ARG].

**Example:**
See Listing 2 on page 11.

```c
int bebop\_TuneMat( bebop\_matrix\_t A\_tunable );
```
Tune the matrix object using all hints and implicit profiling data.

**Parameters:**

**A\_tunable [input/output]**  
Matrix object to tune.  

**Actions and Returns:**

Returns a non-negative status code whose possible values are defined by the constants listed in Table 10 on page 24, or an error code otherwise.

**Error conditions and actions:**

Possible error conditions include providing an invalid matrix [ERR\_BAD\_MATRIX].

**Example:**

See Listing 2 on page 11 and Listing 3 on page 13.

---

### B.5 Permutations

```c
int bebop\_IsMatPermuted( const bebop\_matrix\_t A\_tunable );
```

Checks whether a matrix has been tuned by reordering.

**Parameters:**

**A\_tunable [input]**  
A matrix object corresponding to some matrix \( A \).

**Actions and Returns:**

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

**Error conditions and actions:**

Possible error conditions include providing an invalid matrix [ERR\_BAD\_MATRIX].

**Example:**

See Listing 4 on page 26.

```c
const bebop\_matrix\_t
bebop\_ViewPermutedMat( const bebop\_matrix\_t A\_tunable );
```

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

**Parameters:**

**A\_tunable [input]**  
A matrix object corresponding to some matrix \( A \).
**Actions and Returns:**
Returns a read-only matrix object representing \( \hat{A} \). This return is exactly equal to \texttt{A\_tunable} if the matrix is not reordered, \textit{i.e.}, if \( P_r = P_c = I \), the identity matrix. Returns \texttt{INVALID\_MATRIX} on error.

**Error conditions and actions:**
Possible error conditions include providing an invalid matrix [ERR\_BAD\_MATRIX].

**Example:**
See Listing 4 on page 26.

```c
const bebop\_perm\_t
bebop\_ViewPermutedMat\_RowPerm( const bebop\_matrix\_t A\_tunable );
```

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

**Parameters:**
\texttt{A\_tunable} [input] \hspace{1cm} \texttt{A\_tunable} is valid.
A matrix object corresponding to some matrix \( A \).

**Actions and Returns:**
Returns a read-only permutation object representing \( P_r \). This return is exactly equal to \texttt{PERM\_IDENTITY} if the matrix is not reordered, \textit{i.e.}, if \( P_r = P_c = I \), the identity matrix. Returns \texttt{INVALID\_PERM} on error.

**Error conditions and actions:**
This routine calls the error handler and returns \texttt{INVALID\_MATRIX} if any argument preconditions are not satisfied.

**Example:**
See Listing 4 on page 26.

```c
const bebop\_perm\_t
bebop\_ViewPermutedMat\_ColPerm( const bebop\_matrix\_t A\_tunable );
```

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

**Parameters:**
\texttt{A\_tunable} [input] \hspace{1cm} \texttt{A\_tunable} is valid.
A matrix object corresponding to some matrix \( A \).

**Actions and Returns:**
Returns a read-only permutation object representing \( P_c \). This return is exactly equal to \texttt{PERM\_IDENTITY} if the matrix is not reordered, \textit{i.e.}, if \( P_r = P_c = I \), the identity matrix. Returns \texttt{INVALID\_PERM} on error.

**Error conditions and actions:**
This routine calls the error handler and returns \texttt{INVALID\_MATRIX} if any argument preconditions are not satisfied.
### B.6 Saving and restoring tuning transformations

#### Example:
See Listing 4 on page 26.

```c
int bebob_PermuteVecView( const bebob_perm_t P, bebob_matop_t opP, bebob_vecview_t x_view );
```

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

**Parameters:**
- \( P \) [input] \( P \) is valid.
  - An object corresponding to some permutation, \( P \).
- \( \text{opP} \) [input]
  - Specifies \( \text{op}(P) \).
- \( x\text{view} \) [input/output]
  - The view object corresponding to the (multi)vector \( x \).

**Actions and Returns:**
Permutes the elements of \( x \) and returns 0. On error, returns an error code and leaves \( x\text{view} \) unchanged.

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

#### Example:
See Listing 4 on page 26.

#### B.6 Saving and restoring tuning transformations

```c
char * bebob_GetMatTransforms( const bebob_matrix_t A_tunable );
```

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

**Parameters:**
- \( A\text{tunable} \) [input]
  - \( A\text{tunable} \) is valid.
    - The matrix object to which from which to extract the specified data structure transformations.

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

**Error conditions and actions:**
Possible error codes include providing an invalid matrix \([ERR_BAD_MATRIX]\).
Example:
See Listing 5 on page 27.

```c
int bebop_ApplyMatTransforms( const bebop_matrix_t A_tunable, const char* xforms );
```
Replace the current data structure for a given matrix object with a new data structure specified by
a given string.

**Parameters:**

- **A_tunable** [input]
The matrix object to which to apply the specified data structure transformations.

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

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

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

Example:
See Listing 6 on page 28.

B.7 Error handling

```c
const bebop_errhandler_t
bebop_GetErrorHandler( const bebop_matrix_t A_tunable );
```
Returns a pointer to the current error handling routine for a given matrix object, or the current
global handler if a valid matrix object is not specified.

**Parameters:**

- **A_tunable** [input]
Matrix whose error handler is to be returned, or INVALID_MATRIX for the current global default
handler.

**Actions and Returns:**
Returns a pointer to the current error handler for **A_tunable**, or the current default handler if
**A_tunable == INVALID_MATRIX.** Returns NULL if there is no registered error handler.

```c
const bebop_errhandler_t
bebop_SetErrorHandler( const bebop_matrix_t A_tunable, const bebop_errhandler_t new_handler );
```
Changes the current error handler for a given matrix, or changes the default error handler if no valid matrix object is specified.

**Parameters:**

- **A_tunable [input]**
  Matrix whose error handler is to be returned, or `INVALID_MATRIX` for the current global handler.

- **new_handler [input]**
  A valid error handling routine, or `NULL`.
  Pointer to a new function to handle errors that occur which are associated with matrix `A_tunable`, or a new default error handler if `A_tunable == INVALID_MATRIX`.

**Actions and Returns:**

If `A_tunable` is a valid matrix object, this routine changes the error handler for `A_tunable` to be `new_handler` and returns the previous handler. Otherwise, this routine changes the global error handler to be `new_handler` and returns the previous global default error handler.

```c
void bebop_HandleError_Default( const bebop_matrix_t A_tunable, int error_code,
                                 const char* message, const char* source_filename, unsigned long line_number,
                                 const char* format_string, ... );
```

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

**Parameters:**

- **A_tunable [input]**
  The matrix object being used when the error occurred, or `INVALID_MATRIX` if a matrix object is not applicable.

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

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

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

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

**Actions and Returns:**

This routine dumps a message describing the error to standard error.