

## The Basic Matrix Library (BML) for Quantum Chemistry

Nicolas Bock · Christian F. A. Negre ·  
Susan M. Mniszewski · Jamaludin  
Mohd-Yusof · Bálint Aradi · Jean-Luc  
Fattebert · Daniel Osei-Kuffuor · Timothy  
C. Germann · Anders M. N. Niklasson

Received: date / Accepted: date

---

Nicolas Bock  
Computer, Computational and Statistical Sciences Division, Los Alamos National Laboratory,  
Los Alamos, New Mexico 87545, United States  
SUSE Linux GmbH, Maxfeldstr. 5, 90409 Nürnberg, Germany  
E-mail: nicolasbock@gmail.com

Christian F. A. Negre  
Theoretical Division, Los Alamos National Laboratory, Los Alamos, New Mexico 87545, United States  
E-mail: cnegre@lanl.gov

Susan M. Mniszewski  
Computer, Computational and Statistical Sciences Division, Los Alamos National Laboratory,  
Los Alamos, New Mexico 87545, United States  
E-mail: smm@lanl.gov

Jamaludin Mohd-Yusof  
Computer, Computational and Statistical Sciences Division, Los Alamos National Laboratory,  
Los Alamos, New Mexico 87545, United States  
E-mail: jamal@lanl.gov

Bálint Aradi  
Bremen Center for Computational Materials Science, University of Bremen, Am Fallturm 1,  
28359 Bremen, Germany  
E-mail: aradi@uni-bremen.de

Jean-Luc Fattebert  
Computational Sciences & Engineering Division, Oak Ridge National Laboratory, Oak Ridge,  
TN 37830, United States  
E-mail: fattebertj@ornl.gov

Daniel Osei-Kuffuor  
Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore,  
CA 94551, United States  
E-mail: oseikuffuor1@llnl.gov

Timothy C. Germann  
Theoretical Division, Los Alamos National Laboratory, Los Alamos, New Mexico 87545, United States  
E-mail: tcg@lanl.gov

Anders M. N. Niklasson  
Theoretical Division, Los Alamos National Laboratory, Los Alamos, New Mexico 87545, United States

**Abstract** The basic matrix library package (BML) provides a common application programming interface (API) for linear algebra and matrix functions in C and Fortran for quantum chemistry codes. The BML API is matrix format independent. Currently the dense, CSR and ELLPACK-R sparse matrix data types are available, each with different implementations. We show how the second order spectral projection (SP2) algorithm used to compute the electronic structure of a molecular system represented with a tight-binding (TB) Hamiltonian can be successfully implemented with the aid of this library.

**Keywords** Matrix formats · Matrix-matrix operations · Quantum chemistry packages

*Library title:* Basic Matrix Library (BML)

*Library URL:* <https://zenodo.org/badge/latestdoi/20454/qmmd/bml>

*Licensing provisions:* BSD 3-clause

*Programming language:* Fortran 90 and C

*Compilers:* GNU and INTEL FORTRAN compilers

*Operating system:* Unix/Linux

*Parallelization:* Support for OpenMP and MPI

*External libraries:* LAPACK and BLAS libraries.

*Configuration and building:* CMAKE for building, testing and packaging.

*Purpose of the library:* Matrix operations in several different formats handled through a common API.

## 1 Introduction

The wide variety of computational architectures (multicore, many-core, accelerated), data storage formats (sparse vs. dense), and programming models (distributed, threaded, task-based) renders the implementation, testing, and optimization of scientific algorithms increasingly “overwhelming” [1]. In particular, quantum chemistry packages suffer from this issue because they tend to cover a wide range of physics using algebraic solvers including matrix-matrix operations and are typically computationally very demanding. Consequently, quantum chemical solvers and their underlying matrix technology have been and still are the focus of countless theoretical, computational and pure algorithmic improvements [2]. An advancement of efficient quantum chemistry codes benefits most from a combined effort of both domain scientists in chemistry and physics and computer scientists, which may not always be available.

In most cases, quantum chemistry packages have a common bottleneck. They solve a generalized eigenvalue problem in order to obtain the so called “density matrix”, or the wavefunctions, which characterizes the electronic structure of a molecular system. The computational cost of solving this generalized eigenvalue problem with dense linear algebra scales as  $\mathcal{O}(N^3)$ , where  $N$  is the number of atomic orbitals of the system. One promising approach for computing the density matrix with  $\mathcal{O}(N)$  scaling is by applying the second order spectral projection (SP2) method [3] with

---

States

E-mail: amn@lanl.gov

sparse linear algebra. This algorithm has proven to enable quantum-based molecular dynamics (QMD) simulations of several thousands of atoms on traditional [4, 5] and GPU-accelerated architectures [6, 7].

From a computational point of view, sparse linear algebra tends to be significantly more challenging to optimize than its dense counterpart due to irregular memory storage and access patterns which modern CPUs with their deep memory hierarchies and execution pipelines do not support well. The “optimal” data type and algorithm depends strongly on the physics and physical system studied and potentially changes as advances in sparse linear algebra are made. Quantum chemistry software utilizing sparse linear algebra should therefore not depend on a particular choice of matrix method to enable the agile adaptation of new computational developments. However, with current linear algebra software libraries, the necessary abstraction is not easily achievable and the desired loose coupling of low-level linear algebra and high-level solver technology is challenging to implement.

In this paper, we introduce a Basic Matrix Library (BML) that is matrix storage format and parallel hardware technology agnostic, decoupling the implementation at the lower-level such as storage layout and choice of parallelization from the implementation of higher-order solvers. The library is primarily focused on matrix linear algebra and oriented towards implementation of quantum chemistry packages. Currently, only dense, sparse ELLPACK-R and Compressed Sparse Row (CSR) matrix formats are available but efforts to extend this list are ongoing and will be the subject of upcoming work. Future plans include GPU accelerator implementations for all matrix functions, and the ability to run distributed calculations using MPI. The algorithm implementations are multi-threaded for efficient execution on multi-core single node shared memory architectures. Sharing a similar philosophy, the Matrix Template Library (MTL) [8] serves to provide a general solution of matrix formats and algorithms as a C++ library. The Chebyshev Sparse Solvers (CHESS) library [9] provides a specific sparse matrix format and matrix operations relevant to the use of the Chebyshev polynomial expansion for electronic structure codes. In the same spirit, BML provides matrix formats and algorithms that are most relevant to quantum chemistry codes. Note that the SP2 algorithm is used in this article to illustrate the utility of the BML library. However, the BML library is a general matrix algebra package that can be used for a broad variety of problems.

A large number of available linear matrix libraries [10] is available, however, BML presents unique features that makes it suitable for the development and integration of quantum chemistry kernels. These include API calls that are matrix format agnostic with the possibility of selecting the format at runtime. The density matrix, which is the object that describes the electronic structure of a chemical system can exhibit a wide range of sparsity levels depending on the system [5]. Hence, being able to select the matrix format at runtime is of vital importance. There are lots of solver libraries and programs in the field of quantum chemistry and they all heavily rely on linear matrix algebra. What is not existent as far as we know is a common linear algebra API that could be used by any of the aforementioned programs and libraries, which is what we are introducing here.

The paper is organized as follows: The design and goals of the library are presented first; next, the ELLPACK-R and CSR sparse matrix structure is described together with the ELLPACK-R version of the matrix-matrix multiplication algorithm and its variants. We then give a brief description of the accelerators implementation, including preliminary GPU ELLPACK-R matrix-matrix multiplication

and the MAGMA library. Finally, we describe the implementation of the SP2 algorithm together with an example run in order to show two important points: (1) The fact that the algorithms look as if they were written using a high-level language; and (2) how we can change the choice of matrix format at runtime without changing the code. The last point is essential for benchmarking a computational chemistry algorithm implementation.

## 2 Design goals of the library

The design of high performance data structures and algorithms for dense linear algebra problems is well understood and several optimized implementations of the Basic Linear Algebra Subprograms (BLAS) [11] and the Linear Algebra Package (LAPACK) [12] are available, e.g. Intel Math Kernel Library (MKL) [13], AMD Core Math Library (ACML) [14], AMD Compute Libraries (ACL), and in particular AMD’s cBLAS [15], OpenBLAS [16], GotoBLAS [17], and NVIDIA BLAS Library (cuBLAS) [18].

Optimizing the performance of sparse matrix operations is arguably more challenging because of the irregular nature of data access and work-load. In addition, matrix data distribution depends strongly on the problem domain and work loads and data access are affected by data ordering (Ref. [19] and references therein). A general optimized solution to this problem is not known.

The BML architecture is shown in Figure 1. The core of the library is written in C with a thin Fortran glue layer exposing the API to Fortran 90 applications. Lacking native language support for polymorphism in C, the public matrix data type is a `void` pointer which is resolved at runtime inside the library into appropriate matrix type dependent data structures through a series of nested `switch` statements [20]. This flexibility also enables support of several `float` variants, i.e. currently the library supports dense and ELLPACK-R [21] matrix types and single and double precision real and complex `float` types. Since C does not support generic programming or templates we use a series of preprocessor macros, i.e. `#define` directives [22] to emulate such support, minimizing code duplication. Algorithm implementations are multi-threaded using the Open Multi-Processing (OpenMP) [23] API and multi-threaded BLAS libraries (ex. MKL).

There are two main reasons that C was chosen over other programming languages. These are: (1) Compiler availability: All top 500 supercomputers are running Linux and the Linux kernel requires a C compiler. In addition, new HPC architectures also tend to support Linux and C which is of particular interest to us as part of the Exascale Computing Project (ECP) [24]. (2) Language features: C is a low-level general purpose programming language which can be highly optimized by the compiler. In addition, its flexibility makes it a suitable “lowest common denominator programming language” for interfacing with other languages.

The Fortran programming language itself is very popular among computational quantum chemists as verified by the codes available that are written in this language [25–37]. For this reason, we have provided a Fortran API. Examples of calls to operations throughout the paper are shown in a simplified Fortran syntax. Some examples of the types of matrix algorithms available are shown in Table 1. In most cases, variants are available for all matrix formats.

Table 1: BML Routine examples. The routines are grouped in modules depending on what type of operation they perform. More information can be found in <https://lanl.github.io/bml/API/modules.html>

Algorithm	Examples
Addition	<code>bml_add</code> , <code>bml_add_identity</code>
Allocation	<code>bml_zero_matrix</code> , <code>bml_random_matrix</code> , <code>bml_identity_matrix</code>
Copy	<code>bml_copy</code>
Getters	<code>bml_get_row</code> , <code>bml_get_value</code>
IO	<code>bml_read_matrix</code> , <code>bml_write_matrix</code> , <code>bml_print_matrix</code>
Introspection	<code>bml_get_N</code> , <code>bml_get_precision</code>
Multiply	<code>bml_multiply</code> , <code>bml_multiply_x2</code> , <code>bml_multiply_AB</code>
Norm	<code>bml_fnorm</code>
Normalize	<code>bml_gershgorin</code> , <code>bml_normalize</code>
Scale	<code>bml_scale</code>
Setters	<code>bml_set_row</code> , <code>bml_set_value</code>
Threshold	<code>bml_threshold</code>
Trace	<code>bml_trace</code>
Transpose	<code>bml_transpose</code>

The code is hosted on GitHub [38] and integrated with Travis-CI [39] and `codecov.io` [40] for continuous integration and code coverage analysis. Every commit is tested over a set of compilers and compiler options.

The use of this library is straightforward. In Fortran, for example, the BML main module is included in the application code and a matrix is declared as follows:

```

1  use bml
2  type(bml_matrix_t) :: a

```

Next, it is allocated given the desired type and precision (e.g. dense and double precision). Refer to the manual page on allocation functions for a complete list [38]. For instance,

```

1  call bml_zero_matrix( &
2  & bml_matrix_dense , &
3  & bml_precision_double , &
4  & 100, a)

```

will allocate a dense, double-precision,  $100 \times 100$  matrix initialized to zero. Additional functions allocate special matrices.

- `bml_random_matrix(...)`: Allocates and initializes a random matrix.

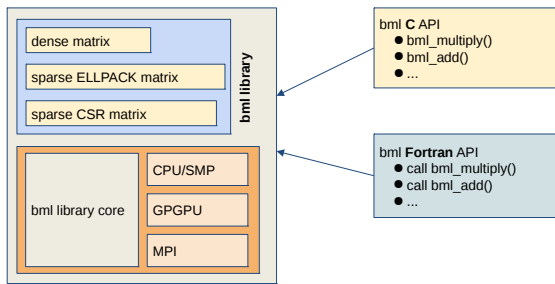


Fig. 1: The Basic Matrix Library (BML) is a set of matrix storage types and algorithms with a common API in C and Fortran. It currently runs on single node multi-core shared memory architectures. Extension to accelerated and distributed memory architectures is under development.

- `bml_identity_matrix(...)`: Allocates and initializes an identity matrix.

A matrix is deallocated by calling:

```
1 call bml_deallocate(a)
```

Automatic deallocation via final subroutines (destructors) in Fortran has intentionally not been implemented yet, as several older but still widely used compilers lack proper support for this Fortran 2003 feature. It is planned, however, to add it when supporting compilers become commonly available on HPC systems.

### 3 The ELLPACK-R sparse matrix format

The dense matrix format is straight forward, requiring a single 2-D array of the numerical values of size  $N \times N$ . In contrast, sparse matrix formats require multiple arrays corresponding to the indices and non-zero values.

The ELLPACK-R format represents a sparse matrix using three arrays: a 2-D real array containing the numerical values, a 2-D integer array containing the column indices, and an integer vector containing the number of non-zero entries per row. The values and indices arrays are both padded with trailing zeros on each row so that each row has a constant offset  $m_{max}$  from the previous row, regardless of occupancy. Thus, the arrays are less compact than other sparse formats, such as CSR, but allow simple strided access for each row and simplified parallelism. There is also no insertion cost compared to CSR because increasing the number of non-zeros in a row does not affect subsequent rows. A schematic representation of the ELLPACK-R format is shown in Figure 2.

By using the BML library, a matrix `a` with ELLPACK-R format can be automatically allocated at runtime by calling `bml_zero_matrix(bml_matrix_ellpack, bml_precision_double, N, M, a)`. In this case we have allocation for  $N$  rows,  $m_{max}$  is set to  $M$  and all entries are set to zero. To start filling up the matrix, we can use the “setter” type of tools. For example, we can set all its rows by using `bml_set_row(a,`

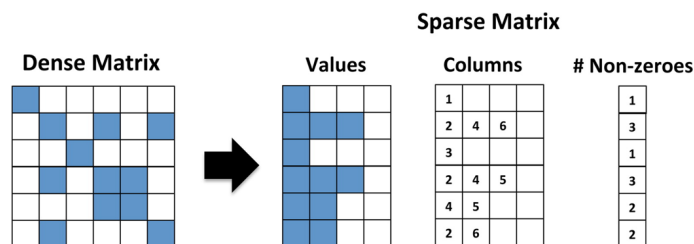


Fig. 2: Sparse matrix ELLPACK-R representation consists of a 2-D array of the non-zero values in each row, a 2-D array of the column indices, and a vector of the number of non-zeros in each row. Reproduced with permission from J. Chem. Theory. Comput., vol. 11, p. 4644 (2015) (reference [4]). Copyright (2017), American Chemical Society

$i$ , `row`) where  $i$  is the row we want to set and `row` is a vector of length  $N$  containing all the values for that particular row.

A global conversion from dense to ELLPACK-R format can be done by using the “import/export” set of routines. For example, if matrix `a` is read or constructed as a 2D dense array, we can call `bml_import_from_dense(bml_type, a, b, threshold, M)`, where in this case `b` results in a matrix with ELLPACK-R format if variable `bml_type` is set to `ellpack`.

It should be noted that processing single matrix elements is not suited for vectorization optimizations since the data is not stored in packed layout. However, avoiding zero matrix elements vs. vectorizing operations on sparse blocks represents a performance trade-off that depends strongly on matrix sparsity. For small atomic basis sets we find that the single matrix element formats presented here outperform blocked formats. For a more detailed analysis see Challacombe [41,19] and Bock *et al.* [42, 43].

#### 4 The CSR sparse matrix format

The CSR storage format is a sparse storage format used in many physics application codes and numerical solver libraries that rely on fast row data accesses to efficiently perform linear algebra operations. Early instances of its use in the literature can be traced back to the mid 1960’s [44,45] and is currently one of the most widely used storage schemes for general purpose sparse linear algebra packages. The most commonly used implementation of the CSR storage scheme utilizes three components: `val`, a floating-point array of the nonzero entries of the matrix taken in row-major order; `col`, an integer array of the corresponding column indexes of the nonzero entries; and `rowptr`, an integer array indexing the position in `val` or `col`, corresponding to the first nonzero entry of each row. It is worth noting that there is an analogous Compressed Sparse Column (CSC) storage scheme, where the storage arrays are populated by traversing the matrix in column-major order. Storing the matrix data in respective contiguous arrays enables fast access for the matrix row data, which benefits the efficient implementation of linear algebra operations. Additional details

about the CSR scheme and variations in its implementation may be found in [46–49], just to mention a few.

The implementation of the CSR storage scheme in BML adopts some modifications to make it flexible and useful for many applications. The key components of the CSR storage data structure in BML are: i) The size of the matrix  $N$  and ii) an array of `sparseRow` struct datatype `data`. Each `sparseRow` datatype contains an integer storing the number of nonzero entries in the row, a floating-point array of the nonzero entries, and an integer array of the corresponding column indexes (see Figure 3).

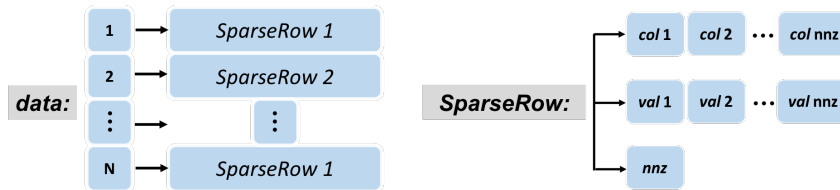


Fig. 3: Schematic of the implementation of the CSR storage structure in BML. (left) variable `data` is an array of pointers to a `sparseRow` struct. (Right) components of each `sparseRow` struct.

The resulting implementation uses the same amount of storage space as the standard approach described earlier. However, this strategy leads to a matrix storage that is extensible and allows for cumulative or incremental matrix assembly. This can be useful for efficient matrix assembly in a distributed parallel framework. For example, in applications using a domain-decomposition framework where each subdomain assembles a matrix from contributions from neighboring subdomains. Two additional variables are available in the CSR storage implementation in BML: an integer array storing the global indexes of the local rows; and a hash table storing (global id, local id) pairs of the matrix rows. Clearly in sequential mode, these additional variables are redundant. However, in parallel, in addition to providing a mapping between local and global variables, they can benefit efficient communication or distribution (and assembly) of matrix data; and are especially useful for situations where the size of the local matrix changes.

## 5 Sparse matrix-matrix multiplication

Dense matrix-matrix multiplication implementations are off-loaded to BLAS Level 3 GEMM routines so that vendor optimized libraries can be used. The implementation of the equivalent sparse matrix operations depend on the sparse matrix format and potentially the CPU architecture target. We find the Gustavson version [50] to work well on multi-core architectures with no dependence on ordering of the non-zero elements. The merge-based variants work well for GPU accelerators and architectures with small cache size but require that the matrix elements remain ordered.

## 5.1 Gustavson algorithm

A sparse matrix-matrix multiplication in ELLPACK-R format, for  $X^2$ , is illustrated in Figure 4. The matrix elements of each row  $i$  of  $X$  are multiplied by the corresponding column vector elements. In this particular example we show the case when  $X$  is symmetric, given that this is a common case encountered in electronic structure calculations. Since  $X$  is symmetric, we multiply by the corresponding rows. These operations are accumulated into a temporary buffer, denoted  $\text{RowBuf}_i$  in Figure 4. The temporary row buffer represents a new row  $i$  of  $X^2$ , which is then stored in the ELLPACK-R format. Since the computation of each row of  $X^2$  is independent of all of the others, this algorithm can be parallelized over rows easily on a shared memory, multi-core node where each thread (or core) operates on a group of one or more rows. Notably, the temporary row buffer is length  $N$ , where  $N \gg m_{max}$ . As each row is added to the row buffer, this can result in random access into the row buffer, which can lead to performance loss due to irregular memory access depending on matrix and available cache sizes. When new non-zero elements are introduced into the row, the indices are appended to the existing index array for that row, which can result in indices being stored out of order.

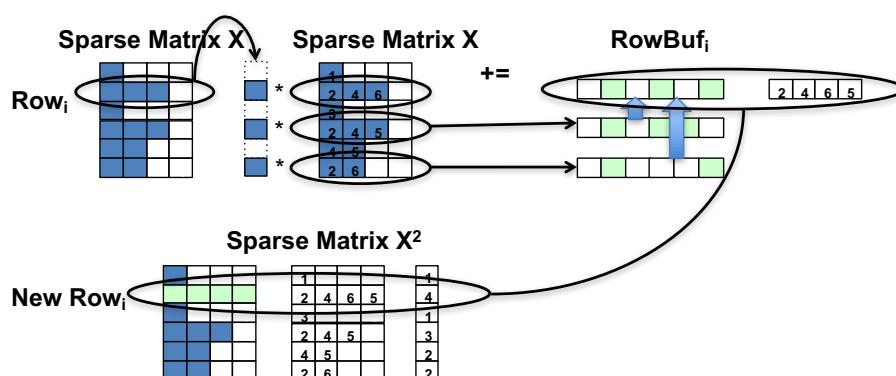


Fig. 4: Gustavson algorithm: Sparse matrix-matrix multiplication is performed in a row-wise fashion. Each non-zero value in a row is multiplied by the non-zero values corresponding to the column index. These values are summed and stored in a temporary row buffer based on their column indices. The row values are thresholded and added to the resulting  $X^2$  matrix. Reproduced and modified with permission from J. Chem. Theory. Comput., vol. 11, p. 4644 (2015) (reference [4]).

The matrix square operation `bml_multiply_x2(x, x2, thresh)` supports thresholding the resulting matrix after the operation where `thresh` is the threshold. This thresholding keeps all the elements that have absolute value greater than `thresh`, and removes all the remaining elements below the threshold.

## 5.2 Merge-based algorithm

Figure 5 illustrates a merge-based matrix-matrix multiply working on a matrix in the ELLPACK-R format. Since the row elements, when imported from a dense matrix, are stored in order, we can compute the result of combining any two rows by merging those rows. Thus, the operation can complete in  $\log(m)$  stages, where  $m$  is the number of non-zeros in the row. The total memory required is  $\mathcal{O}(m^2)$  but memory access patterns are regular and only require contiguous blocks of length  $m$ . Thus, this algorithm can perform well on architectures with small caches, such as GPUs, particularly when  $N$  is large and the matrix has high bandwidth.

Some architectures may not have sufficient cache to accommodate all the temporary arrays required by the previous implementation. An alternative formulation is presented in Figure 6, where the pairwise merge is replaced by merging successive rows of the result with the new row. This results in  $\mathcal{O}(m)$  storage, but requires  $m - 1$  stages.

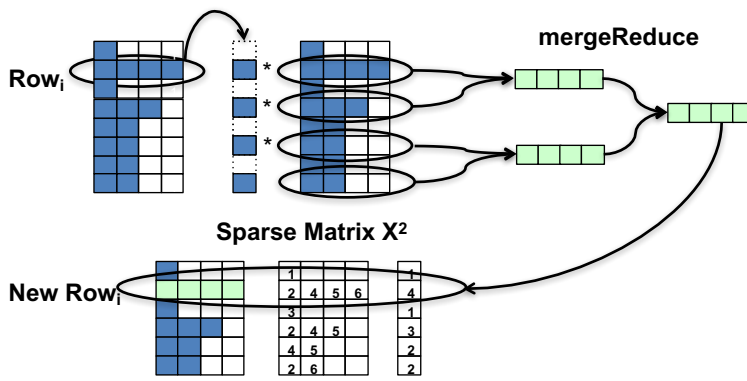


Fig. 5: Merge-based algorithm: Sparse matrix-matrix multiplication is performed in a row-wise fashion. Each non-zero value in a row is multiplied by the non-zero values corresponding to the column index. The resulting pairs of rows are recursively merged into the new row. The row values are thresholded and added to the resulting  $X^2$  matrix.

## 6 Accelerators

Within the definition of accelerators are included GPUs as well as any potential future architectures. We are working on implementing this in a variety of ways, including native OpenMP offload functionality as well as by allowing integration with existing libraries such as CUDA/CUBLAS/CUSPARSE and MAGMA. As with all discrete accelerators, the movement of data between host and device is a bottleneck to be avoided. Once moved to the device, all computations are performed there, and only final results are returned to the CPU. We therefore choose to restrict the data offload to the allocation subroutines of BML, and including an ac-

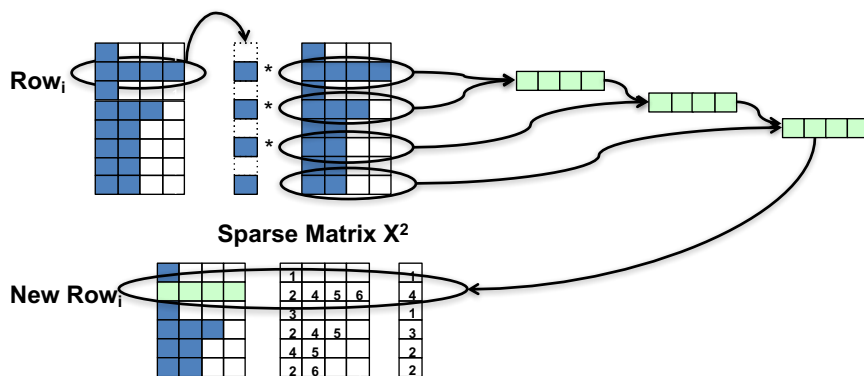


Fig. 6: Low-storage merge-based algorithm: Sparse matrix-matrix multiplication is performed in a row-wise fashion. Each non-zero value in a row is multiplied by the non-zero values corresponding to the column index. The resulting rows are successively merged into the new row. The row values are thresholded and added to the resulting  $X^2$  matrix.

celeration type with the `bml_device_type` specification. Once a matrix of appropriate type is instantiated, all subsequent computations will be performed on the device. Only when the results are exported from the BML library will they be returned to the host processor. Syntactically, this will involve the addition of an argument to denote the accelerator mode when BML objects are initialized. Once initialized, the `bml_device_type` will include the accelerator mode, so subsequent operations on those objects will be transparent. For the case of `omp_offload`, allocation looks like: `bml_matrix_t *A = bml_zero_matrix(ellpack, double_real, n, m, sequential, omp_offload);` All other calls to the library would remain unchanged. ...

### 6.1 Merge-based preliminary results on GPU

An initial implementation of the merge-based method was presented at GTC 2015 [51]. Figure 7 shows the performance of the merge-based algorithm compared to the NVIDIA cuSparse library version on an NVIDIA K40 GPU. While the new algorithm only marginally outperforms cuSparse for ordered systems (which result in matrices with small bandwidth) there is a significant improvement in cases where the system is disordered (i.e. matrices have high bandwidth), even for relatively small matrices. The relative improvement of the new method increases for increasing matrix size and, as we expect, for matrices with lower sparsity (more non-zeros per row) due to the ability to keep the working set in shared memory. A revisit of these tests with the latest GPU hardware and software using larger matrices will be addressed in future work.

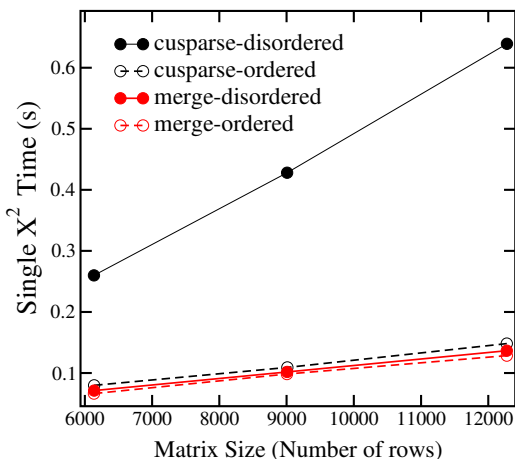


Fig. 7: Preliminary results comparing the merge-based sparse matrix-matrix multiply algorithm to cuSparse, on an NVIDIA K40 GPU using CUDA 6.5. The new method outperforms cuSparse by a factor of 4 to 5 even for modest matrix sizes in disordered cases.

## 6.2 MAGMA accelerator

MAGMA is a dense linear algebra library similar to LAPACK, but designed to run on GPUs [52]. Since it implements a large fraction of the functionalities provided by BML and performs very well, it is natural to use it directly instead of reimplementing similar code. BML does not assume MAGMA is available on any given platform. But if it is and the user decides to make use of it, dense linear code in BML is replaced by GPU memory allocations and MAGMA calls at compile time. This is done with conditional groups (`#ifdef...#else...#endif`) activated if a specific macro is defined. In that case, the BML code is reduced to a thin layer wrapper around MAGMA functions. Performance is thus very close to direct calls to MAGMA functions. An illustration of the performance obtained on a GPU for a matrix-matrix multiplication is shown in Fig. 8. Note that not all functionalities supported by BML are implemented in MAGMA, and thus a specific implementation is needed for these. On the other hand, while some functions are currently supported by BML for convenience, and are also supported for dense linear algebra built with MAGMA — such as accessors to the individual matrix elements — they may lead to a very poor performance if an algorithm was to be implemented in the user’s code, and rely heavily on these. While BML does not enforce a strict data encapsulation in that sense, it is clear that computationally expensive operations should be implemented within BML if one wants to take advantage of GPU accelerators.

## 7 Implementation of the SP2 algorithm using BML

In quantum chemistry, the density matrix  $\mathbf{P}$  characterizes the electronic structure of a molecular system since with this object, any expectation value or average property

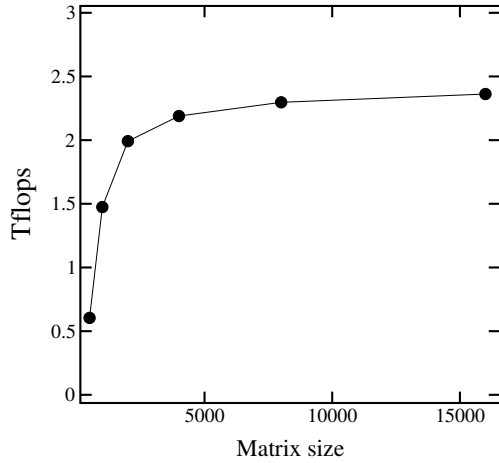


Fig. 8: Measured performance for a BML dense matrix-matrix multiplication in double precision based on MAGMA. Platform: single NVIDIA Tesla P100 GPU on Summitdev at the Oak Ridge Leadership Computing Facility.

$\langle A \rangle$  can be easily computed as  $\text{Tr}(\mathbf{P}A)$ , where  $A$  is the matrix representation of any general operator [53].

The density matrix can be computed as the Fermi function of the Hamiltonian matrix  $H$  as  $\mathbf{P} = f(H)$ . When expressed in an orthogonal basis of localized atomic orbitals the latter equation reads:  $\mathbf{P}_{ij} = [\mathbf{C}f(\epsilon)\mathbf{C}^\dagger]_{ij}$ , where matrices  $\epsilon$  and  $\mathbf{C}$  are the matrices containing the eigenvalues  $\epsilon_i$  and eigenvectors  $C_i$  respectively [53]. This implies that, in order to construct matrix  $\mathbf{P}$  we need first to solve the eigenvalue problem  $\mathbf{H}\mathbf{C} = \mathbf{C}\epsilon$  for which the computational cost scales as  $\mathcal{O}(N^3)$ .

The SP2 method allows us to obtain  $\mathbf{P}$  from  $\mathbf{H}$  directly without the need of performing a matrix diagonalization. It is based on a recursive expansion of the Fermi operator. At zero electronic the Fermi operator can be written as follows:

$$\mathbf{P} = \Theta[\mu\mathbf{I} - \mathbf{H}] \quad (1)$$

where  $\mathbf{I}$  is the identity matrix, the function  $\Theta(\cdot)$  is the Heaviside step function and  $\mu$  is the chemical potential determined such that the trace of  $\mathbf{P}$  is the number of occupied states,  $N_{occ}$ , i.e.  $\text{Tr}[\mathbf{P}] = N_{occ}$ . By writing a recursive expansion of  $\Theta$ , we have:

$$\Theta[\mu\mathbf{I} - \mathbf{H}] = \lim_{i \rightarrow \infty} f_i(f_{i-1}(\dots f_0(\mathbf{X}_0))) \quad (2)$$

where  $\mathbf{X}_0$  is the initial member of the sequence which is computed as:

$$\mathbf{X}_0 = \frac{\epsilon_{\max}\mathbf{I} - \mathbf{H}}{\epsilon_{\max} - \epsilon_{\min}} \quad (3)$$

where  $\epsilon_{\min}$  and  $\epsilon_{\max}$  are spectral bound estimates, e.g. calculated using the Gershgorin circle theorem [54]. Function  $f_i$  are applied as follows:

$$f_i(\mathbf{X}_i) = \begin{cases} 2\mathbf{X}_i - \mathbf{X}_i^2, & \text{if } \text{Tr}(\mathbf{X}) \leq N_{occ} \\ \mathbf{X}_i^2, & \text{if } \text{Tr}(\mathbf{X}) > N_{occ} \end{cases} \quad (4)$$

Alternatively, the functions  $f_i$  are chosen to minimize the difference between the trace of  $f_i(\mathbf{X}_i)$  and  $N_{occ}$ , which avoids potential instabilities under approximate numerically thresholded calculations. These recursive functions are applied until  $\text{Tr}(\mathbf{X}) \simeq N_{occ}$  for when we finally have  $\mathbf{P} = \mathbf{X}$ . A pseudocode for the SP2 algorithm is given in Algorithm 1, whereas the simplified Fortran code is given in Algorithm 2.

---

**Algorithm 1** Pseudocode for the SP2 algorithm.

---

```

procedure SP2(tol, H, P,  $N_{occ}$ )
  Estimate  $\epsilon_{\min}$  and  $\epsilon_{\max}$  from H
   $\mathbf{X} = (\epsilon_{\max}\mathbf{I} - \mathbf{H})/(\epsilon_{\max} - \epsilon_{\min})$ 
  TraceX = Tr[X]
  for  $i = 1 : i_{\max}$  do
    TraceXold = TraceX
     $\mathbf{X}_{\text{tmp}} = \mathbf{X}^2$ 
    if  $(\text{TraceX} - N_{occ}) \leq 0$  then
       $\mathbf{X} = 2\mathbf{X} - \mathbf{X}_{\text{tmp}}$ 
    else
       $\mathbf{X} = \mathbf{X}_{\text{tmp}}$ 
    end if
    if  $|\text{TraceX} - N_{occ}| \leq \text{tol}$  then
      break
    end if
  end for
  P = X
end procedure

```

---

Algorithm 2 receives a previously allocated Hamiltonian matrix `h_bml` and returns the density matrix `p_bml`. The original Hamiltonian matrix is left unchanged for future processing. Algorithm 2 shows several useful functions and routines implemented in the BML library. We will go through each of the routines that are used. An important point to note is that Algorithm 2 looks as if it was written in the highest-level language possible, where every single operation can be easily mapped to Algorithm 1 [55].

At the beginning of the routine the `use bml` statement enables the use of the library and all its routines. The declaration of the variables are omitted for simplicity reasons but the important point to take into account is that the BML matrices have to be declared as explained in Section 2. The Hamiltonian for example, is declared as: `type(bml_matrix_t) :: h_bml`. The first operation performed inside the routine is to get the dimension of the matrix which corresponds to the number of atomic orbitals. This is done using the `bml_get_N` function. The Gershgorin circle theorem [54] routine (`bml_gershgorin`) calculates estimates of the lower and upper spectral bounds,  $\epsilon_{\min}$  and  $\epsilon_{\max}$  for a given Hamiltonian matrix. Using the BML routines for scaling (`bml_scale`) and adding an identity matrix (`bml_add_identity`) the first argument for the recursive loop is obtained.

Within the loop we identify several operations. The  $X^2$  matrix-matrix multiply operation, the trace operation, the copy assignment operation, and the addition operation (`bml_multiply_x2`, `bml_trace`, `bml_copy`, `bml_add`).

The algorithm finishes when  $(\text{abs}(N_{occ} - \text{trx}) < \text{tol})$ . We note that the temporary matrix `x2_bml` is allocated at the beginning using `bml_zero_matrix` with some

characteristics such as the dimension `Norb` and matrix format `bml_type`. At the end of the algorithm `x2_bml` is deallocated using `bml_deallocate`. It is important to note that all the matrices must be properly deallocated to avoid memory problems.

---

**Algorithm 2** Abbreviated Fortran code for the SP2 algorithm using BML routines. Given `h_bml`, `Nocc` and some control variables (`Control_vars`), a matrix `p_bml` is computed.

---

```

subroutine SP2_BML(Control_vars, h_bml, p_bml, Nocc)
  use bml
  Declare variables ...
  Initialize variables ...
  Norb= bml_get_N(h_bml)
  call bml_copy(h_bml, p_bml)
  call bml_gershgorin(p_bml, Gersh)
  emin= Gersh(1); emax= Gersh(2);
  call bml_scale(-1.0/(emax-emin), p_bml)
  GershFact= emax/(emax-emin)
  call bml_add_identity(p_bml, GershFact, thresh)
  call bml_zero_matrix(bml_type, bml_element_real
  &, precision, Norb, x2_bml)
  do iter = 1, imax
    trx = bml_trace(p_bml)
    call bml_multiply_x2(p_bml, x2_bml, thresh)
    if(trx- Nocc <= 0.0) then
      call bml_add(2.0,p_bml,-1.0,x2_bml)
    else
      call bml_copy(x2_bml, p_bml)
    end if
    if (abs(Nocc-trx) < tol) exit
    if(iter == maxiter) stop
  end do
  call bml_deallocate(x2_bml)
end subroutine

```

---

We have tested this implementation of the SP2 algorithm with Hamiltonian matrices computed from a series of water boxes of increasing sizes. Each of these Hamiltonians  $\mathbf{H}$  are constructed with the self-consistent-charge density functional based tight-binding code (DFTB+) [56] from the *mio.org* Slater-Koster set of parameters [57,58]. These Hamiltonians were previously orthogonalized applying the Löwdin factorization method [59,5]. Tolerance for the SP2 (parameter `tol`) method was set to  $10^{-7}$ . The matrix threshold for eliminating values close to zero for the ELLPACK (parameter `thresh`) format was set to  $10^{-5}$ .

Performance of the SP2 method using the previously computed Hamiltonians is shown in Figure 9. The wall-clock time for a single density matrix construction is shown for water systems of increasing size (300–2100 atoms). Matrix operation algorithms at the BML level are specifically tailored for each particular matrix format, hence, completely different routines are executed for dense (black curve) vs. ELLPACK-R (red curve) formats. An optimized implementation of the BLAS Level 3 Double precision General Matrix-Matrix multiplication (DGEMM) is used

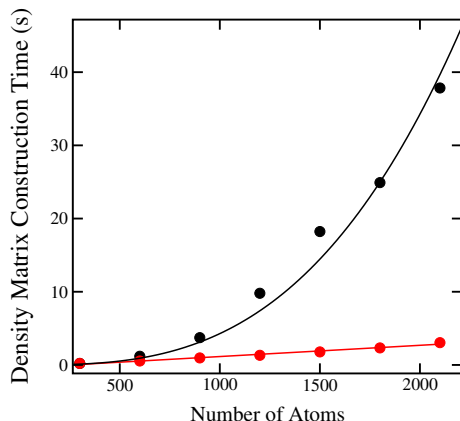


Fig. 9: SP2 execution times showing performance differences between the dense (black curve) and ELLPACK-R (red curve) matrix formats.

for dense as compared to a modified Gustavson algorithm [50] that takes advantage of symmetry and sparsity used for ELLPACK-R [4]. The format is selectable at run time by modifying the parameter `bml_type`. The bottleneck of the SP2 algorithm is the matrix-matrix multiply. Compute time is greatly reduced when sparse methods are applicable. All the operations were threaded using 8 cores/threads on a 2.70GHz Intel i7-4800MQ processor. Figure 9 (red curve) shows linear scaling for the ELLPACK-R SP2 method. In comparison, when “dense” format is selected, the SP2 method scales as  $\mathcal{O}(N^3)$  (black curve of Figure 9). The ELLPACK-R matrix-matrix multiply is implemented as explained in Section 5.1, while for dense, a BLAS Level 3 DGEMM call is made. With this example, we show that the scaling of the SP2 algorithm with the system size becomes linear when switching from dense to ELLPACK. This format change is done at runtime by setting `bml_type` to `ellpack`.

The characteristics of the chemical system will strongly depend on the chemical elements conforming the system i.e. if the system is metallic, semiconductor, molecular, etc. The level of sparsity varies across these different characteristics and in consequence, the total memory consumption and workload. The domain scientist running a Quantum Chemistry calculation is faced with decisions about matrix format and architecture at runtime depending on the system. This is the reason for the high-flexibility offered by this library in terms of matrix formats and devices. The ELLPACK format should be used for non-metallic systems, such as insulators and bio-systems, where the density matrix is less than 50% sparse. The ELLPACK format can be used for dense systems but is not recommended, due to the loss of efficiency, resulting in larger memory usage and reduced performance.

Using BML in a code allows for one version in many cases, instead of different variants. Instead of having a lot of “if” statements depending on the matrix format, the selection is made at execution time. We have shown how a linear algebra based quantum chemistry solver such as the SP2 algorithm can be easily coded up in a high-level style where matrix formats and matrix operations are performed at the library level. Switching from one matrix format to the other can be done at execution time and the performance of the whole code can be easily tested. An available BML

version of the SP2 algorithm can be found in the PROGRESS library [60]. The PROGRESS library offers a collection of BML based quantum chemistry solvers including different versions of the SP2; Chebyshev kernel polynomial method; inverse overlap matrix calculation method; density matrix response; etc.

## 8 Conclusion

We have introduced the BML library to perform different matrix operations where the format can be decided at runtime. The freedom of choosing any matrix format type allows a user to focus on the development of a high-level solver without having to care about the low-level implementation. We are continuously extending the supported formats, adding more functionalities as well as targeting emerging architectures. This library has been developed for ease of use in writing quantum chemistry programs. The simplicity of bml, together with the fact that it is written in C makes it suitable for straightforward integration in C/C++ and FORTRAN codes (usually the preferred languages for quantum chemistry codes). The fact that it has high flexibility in terms of compilation (different cmake building options including a “non-library-dependence mode”) makes it particularly attractive for quantum chemistry kernels intended to run on emerging exascale architectures.

## Acknowledgement

This article was approved for unlimited release with the following LA-UR number: 'LA-UR-17-29481'. This library was developed using funding from: (1) Basic Energy Sciences (LANL2014E8AN) and the Laboratory Directed Research and Development Program of Los Alamos National Laboratory. To tests these developments we used resources provided by the Los Alamos National Laboratory Institutional Computing Program, which is supported by the U.S. Department of Energy National Nuclear Security Administration. (2) Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative. First and second authors have equally contributed to the manuscript.

## References

1. Z. Merali, Computational science: Error, why scientific programming does not compute, *Nature* 467 (7317) (2010) 775–777. doi:10.1038/467775a.  
URL <http://dx.doi.org/10.1038/467775a>
2. D. S. Watkins, *Fundamentals of matrix computations*, third edition, John Wiley & Sons, 2010.
3. A. M. N. Niklasson, Expansion algorithm for the density matrix, *Phys. Rev. B* 66 (2002) 155115.
4. S. M. Mniszewski, M. J. Cawkwell, J. Mohd-Yusof, N. Bock, T. C. Germann, A. M. N. Niklasson, Parallel linear scaling calculation of the density matrix in electronic structure theory, *J. Chem. Theory Comput.* 11 (10) (2015) 4644 – 4654.

5. C. F. A. Negre, S. M. Mniszewski, M. J. Cawkwell, N. Bock, M. E. Wall, A. M. N. Niklasson, Recursive factorization of the inverse overlap matrix in linear-scaling quantum molecular dynamics simulations, *J. Chem. Theory Comput.* 12 (7) (2016) 3063–3073. doi:10.1021/acs.jctc.6b00154.
6. M. J. Cawkwell, E. J. Sanville, S. M. Mniszewski, A. M. N. Niklasson, Computing the density matrix in electronic structure theory on graphics processing units, *J. Chem. Theory Comput.* 8 (11) (2012) 4094–4101.
7. M. J. Cawkwell, M. A. Wood, A. M. N. Niklasson, S. M. Mniszewski, Computation of the density matrix in electronic structure theory in parallel on multiple graphics processing units, *J. Chem. Theory Comput.* 10 (12) (2014) 5391–5396.
8. SimuNova, Matrix template library (mtl) (2016).  
URL <http://www.simunova.com/home>
9. S. Mohr, W. Dawson, M. Wagner, D. Caliste, T. Nakajima, L. Genovese, Efficient computation of sparse matrix functions for large-scale electronic structure calculations: The chess library, *J. Chem. Theory Comput.* 13 (2017) 4684–4698.
10. Freely available software for linear algebra (2018).  
URL <http://www.netlib.org/utk/people/JackDongarra/la-sw.html>
11. M. J. Daydé, I. S. Duff, A block implementation of level 3 blas for risc processors, to appear (1995).
12. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, SIAM, release 2.0 Edition (1994).
13. Intel, Intel math kernel library (intel mkl) (2017).  
URL <https://software.intel.com/en-us/intel-mkl>
14. AMD, Amd core math library (acml) (2017).  
URL <http://developer.amd.com/tools-and-sdks/archive/amd-core-math-library-acml/>
15. AMD, Amd core math library clblas (2017).  
URL <https://github.com/clMathLibraries/clBLAS>
16. OpenBLAS, Openblas library (2017).  
URL <http://www.openblas.net/>
17. TACC, Gotoblas2 library (2017).  
URL <https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2>
18. Nvidia, cublas (2014).  
URL <http://developer.nvidia.com/cuBLAS>
19. M. Challacombe, A general parallel sparse-blocked matrix multiply for linear scaling scf theory, *Comput. Phys. Comm.* 128 (1) (2000) 93 – 107. doi:[https://doi.org/10.1016/S0010-4655\(00\)00074-6](https://doi.org/10.1016/S0010-4655(00)00074-6).  
URL <http://www.sciencedirect.com/science/article/pii/S0010465500000746>
20. cppreference.com, Switch statement (2017).  
URL <http://en.cppreference.com/w/c/language/switch>
21. F. Vasquez, G. Ortega, J. J. Fernandez, E. Garzon, Improving the performance of the sparse matrix vector product with gpus, 10th IEEE International Conference on Computer and Information Technology (2010) 1146–1151.
22. cppreference.com, Preprocessor (2017).  
URL <http://en.cppreference.com/w/cpp/preprocessor/replace>
23. O. A. R. Board, Openmp (2014).  
URL <http://openmp.org>
24. The ecp homepage (2017).  
URL <https://exascaleproject.org/>
25. The dftb+ homepage (2017).  
URL <https://www.dftbplus.org/>
26. The siesta homepage.  
URL <https://departments.icmab.es/leem/siesta/>
27. Cp2k project homepage (2017).  
URL <http://cp2k.berlios.de>
28. The adf modeling suite homepage (2017).  
URL <https://www.scm.com/>
29. The cpmd homepage (2017).  
URL <http://www.cpmc.org/>

30. The dacapo homepage (2017).  
URL <https://wiki.fysik.dtu.dk/dacapo>
31. The gamess homepage (2017).  
URL <http://www.msg.chem.iastate.edu/gamess/>
32. The gaussian homepage (2017).  
URL <http://gaussian.com/>
33. Bigdft homepage (2017).  
URL [http://bigdft.org/Wiki/index.php?title=BigDFT\\_website](http://bigdft.org/Wiki/index.php?title=BigDFT_website)
34. The vasp homepage (2017).  
URL <http://www.vasp.at/>
35. The turbomole homepage (2017).  
URL <http://www.cosmologic.de/turbomole/home.html>
36. The mopac homepage (2017).  
URL <http://openmopac.net/>
37. The jaguar homepage (2017).  
URL <https://www.schrodinger.com/jaguar>
38. B. Aradi, N. Bock, S. M. Mniszewski, J. Mohd-Yusof, C. Negre, The basic matrix library manual (2016).  
URL <https://qcmd.github.io>
39. Travis-CI, Travis-CI (2017).  
URL <https://travis-ci.org/>
40. Codecov, Codecov (2017).  
URL <https://codecov.io/>
41. M. Challacombe, A simplified density matrix minimization for linear scaling self-consistent field theory, *J. Chem. Phys.* 110 (5) (1999) 2332–2342.
42. M. Challacombe, N. Bock, Fast multiplication of matrices with decay, *CoRR* abs/1011.3534. arXiv:1011.3534.  
URL <http://arxiv.org/abs/1011.3534>
43. N. Bock, M. Challacombe, An optimized sparse approximate matrix multiply for matrices with decay 35 (1) C72–C98. arXiv:<https://doi.org/10.1137/120870761>, doi:10.1137/120870761.  
URL <https://doi.org/10.1137/120870761>
44. N. Sato, W. Tinney, Techniques for exploiting the sparsity or the network admittance matrix, *IEEE Transactions on Power Apparatus and Systems* 82 (69) (1963) 944–950.
45. W. F. Tinney, J. W. Walker, Direct solutions of sparse network equations by optimally ordered triangular factorization, *Proceedings of the IEEE* 55 (11) (1967) 1801–1809. doi:10.1109/PROC.1967.6011.
46. Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd Edition, Society for Industrial and Applied Mathematics, 2003. arXiv:<https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003>, doi:10.1137/1.9780898718003.  
URL <https://epubs.siam.org/doi/abs/10.1137/1.9780898718003>
47. A. George, J. W. Liu, *Computer Solution of Large Sparse Positive Definite*, Prentice Hall Professional Technical Reference, 1981.
48. T. Davis, *Direct Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, 2006. arXiv:<https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881>, doi:10.1137/1.9780898718881.  
URL <https://epubs.siam.org/doi/abs/10.1137/1.9780898718881>
49. G. H. Golub, V. L. C. F., *Matrix computations*, Johns Hopkins Univ. Press, 2007.
50. F. G. Gustavson, Two fast algorithms for sparse matrices: Multiplication and permuted transposition, *ACM Trans. Math. Soft.* 4 (3) (1978) 250–269.
51. J. Mohd-Yusof, N. Sakharnykh, S. M. Mniszewski, M. J. Cawkwell, N. Bock, T. C. Germann, A. M. N. Niklasson, Fast sparse matrix multiplication for QMD using parallel merge, *GPU Technology Conference* (2015).
52. J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, Accelerating numerical dense linear algebra calculations with gpus, *Numerical Computations with GPUs* (2014) 1–26.
53. J. J. Sakurai, *Modern Quantum Mechanics*, Addison Wesley, 1994.
54. G. Golub, C. F. van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1996.

55. G. Wilson, D. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, et al., Best practices for scientific computing, *PLoS Biol* 12 (1) (2014) e1001745.
56. B. Aradi, B. Hourahine, T. Frauenheim, Dftb+, a sparse matrix-based implementation of the dftb method, *J. Phys. Chem. A* 111 (26) (2007) 5678–5684, PMID: 17567110. doi:10.1021/jp070186p.  
URL <http://www.dftbplus.org>
57. J. C. Slater, G. F. Koster, Simplified lcao method for the periodic potential problem, *Phys. Rev.* 94 (1954) 1498–1524. doi:10.1103/PhysRev.94.1498.  
URL <http://link.aps.org/doi/10.1103/PhysRev.94.1498>
58. M. Elstner, D. Porezag, G. Jungnickel, J. Elsner, M. Haugk, T. Frauenheim, S. Suhai, G. Seifert, Self-consistent-charge density-functional tight-binding method for simulations of complex materials properties, *Phys. Rev. B* 58 (1998) 7260–7268. doi:10.1103/PhysRevB.58.7260.  
URL <http://link.aps.org/doi/10.1103/PhysRevB.58.7260>
59. P. O. Löwdin, Quantum theory of cohesive properties of solids, *Adv. Phys.* 5 (1956) 3–164.
60. A. M. Niklasson, S. M. Mniszewski, C. F. A. Negre, M. E. Wall, M. J. Cawkwell, N. Bock, PROGRESS version 1.0 (2016).  
URL <https://github.com/lanl/qmd-progress>