User's Guide for SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package (with optional GPU acceleration)

Timothy A. Davis*, Sencer Nuri Yeralan, Sanjay Ranka, Wissam Sid-Lakhdar

VERSION 4.2.1, Sept 18, 2023

Abstract

SuiteSparseQR is an implementation of the multifrontal sparse QR factorization method. Parallelism is exploited both in the BLAS and across different frontal matrices using Intel's Threading Building Blocks, a shared-memory programming model for modern multicore architectures. It can obtain a substantial fraction of the theoretical peak performance of a multicore computer. The package is written in C++ with user interfaces for MATLAB, C, and C++. Both real and complex sparse matrices are supported.

1 Introduction

The algorithms used in SuiteSparseQR are discussed in a companion paper, [7], and an overview of how to use the software is given in [6]. This document gives detailed information on the installation and use of SuiteSparseQR.

2 Using SuiteSparseQR in MATLAB

The simplest way to use SuiteSparseQR is via MATLAB. Its syntax includes every feature of the MATLAB `qr` in version R2009a and earlier [12], plus additional features not available in MATLAB. It is also a replacement for `x=A\b` for least-squares problems and underdetermined systems. In addition to substantial gains in performance (10x to 100x is not uncommon, up

*email: DrTimothyAldenDavis@gmail.com. http://www.suitesparse.com. Portions of this work were supported by the National Science Foundation, under grants 0203270, 0620286, and 0619080.

to 10,000x has been observed), SuiteSparseQR adds new capabilities that are not present in MATLAB. For example, it provides an efficient method for finding the minimum 2-norm solution to an underdetermined system.

2.1 Installing SuiteSparseQR for use in MATLAB

All packages in SuiteSparse, including SuiteSparseQR and the codes it relies on (AMD, COLAMD, CHOLMOD, METIS, CCAMD, and CCOLAMD) are compiled with a single command typed into the MATLAB Command Window. SuiteSparseQR uses the LAPACK and BLAS libraries provided with MATLAB; you do not need to do anything to use these. Below are step-by-step instructions for compiling all of SuiteSparse (including SuiteSparseQR), and optional instructions on using METIS.

2.1.1 Now you're ready to compile (on any operating system)

Type these commands in the MATLAB window:

```
cd SuiteSparse
SuiteSparse_install
```

You will be asked if you want to run some demos. I recommend that you do this to ensure your functions have been installed correctly. Next type the command

```
pathtool
```

and examine your MATLAB path. The various SuiteSparse directories have been placed in your path. Click "save" to save this path for future MATLAB sessions. If this fails, you do not have permission to modify the `pathdef.m` file (it is shared by all users).

SuiteSparse now has a `SuiteSparse_paths.m` that you can add to your `startup.m`. An alternative is to type the command:

```
path
```

and cut-and-paste the paths displayed there into your own `startup.m` file, prepending the command `addpath` to each line.

Your `startup.m` file should appear in the directory in which MATLAB starts. Failing that, every time you start MATLAB, find your `startup.m` file and run it. For more help, type `doc startup` in MATLAB.

The `SuiteSparse_install` script works on any version of MATLAB (Linux/Unix, Mac, or Windows) if you have a C++ compiler. The install script will detect if you have placed the METIS directory in the right place, and will compile it for use with SuiteSparseQR if it finds it there. Otherwise METIS will be skipped (the install script will tell you if it finds METIS or not).

2.2 Functions provided to the MATLAB user

Three primary functions are available:

1. `spqr`, a replacement for the MATLAB `qr`
2. `spqr_solve`, a replacement for `x=A\b` when `A` is sparse and rectangular. It works for the square case, too, but `x=A\b` will be faster (using LU or Cholesky factorization). `spqr_solve` is a good method for ill-conditioned or rank-deficient square matrices, however.
3. `spqr_qmult`, which multiplies `Q` (stored in Householder vector form) times a matrix `x`.

Their syntax is described below in the table below. The permutation `P` is chosen to reduce fill-in and to return `R` in upper trapezoidal form if `A` is estimated to have less than full rank. The `opts` parameter provides non-default options (refer to the next section). The output `Q` can be optionally returned in Householder form, which is far sparser than returning `Q` as a sparse matrix.

`R = spqr (A)`	Q-less QR factorization
`R = spqr (A,0)`	economy variant (`size(R,1) = min(m,n)`)
`R = spqr (A,opts)`	as above, with non-default options
`[Q,R] = spqr (A)`	`A=Q*R` factorization
`[Q,R] = spqr (A,0)`	economy variant (`size(Q,2) = size(R,1) = min(m,n)`)
`[Q,R] = spqr (A,opts)`	`A=Q*R`, with non-default options
`[Q,R,P] = spqr (A)`	`A*P=Q*R` where P reduces fill-in
`[Q,R,P] = spqr (A,0)`	economy variant (`size(Q,2) = size(R,1) = min(m,n)`)
`[Q,R,P] = spqr (A,opts)`	as above, with non-default options
`[C,R] = spqr (A,B)`	as `R=spqr(A)`, also returns `C=Q'*B`
`[C,R] = spqr (A,B,0)`	economy variant (`size(C,1) = size(R,1) = min(m,n)`)
`[C,R] = spqr (A,B,opts)`	as above, with non-default options
`[C,R,P] = spqr (A,B)`	as `R=spqr(A*P)`, also returns `C=Q'*B`
`[C,R,P] = spqr (A,B,0)`	economy variant (`size(C,1) = size(R,1) = min(m,n)`)
`[C,R,P] = spqr (A,B,opts)`	as above, with non-default options
`x = spqr_solve (A,B)`	`x=A\B`
`[x,info] = spqr_solve (A,B,opts)`	as above, with statistics and non-default parameters
`Y = spqr_qmult (Q,X,k)`	computes `Q'*X`, `Q*X`, `X*Q'`, or `X*Q` (selected with `k`)

2.3 The `opts` parameter

The `opts` struct provides control over non-default parameters for SuiteSparseQR. Entries not present in `opts` are set to their defaults.

- `opts.tol`: columns that have 2-norm `<= opts.tol` are treated as zero. The default is `20*(m+n)*eps*sqrt(max(diag(A'*A)))` where `[m n]=size(A)`.
- `opts.econ`: number of rows of `R` and columns of `Q` to return. The default is `m`. Using `n` gives the standard economy form (as in the MATLAB `qr(A,0)`). A value less than the estimated rank `r` is set to `r`, so `opts.econ=0` gives the "rank-sized" factorization, where `size(R,1)==nnz(diag(R))==r`.

- `opts.ordering`: a string describing which column ordering method to use. Let `[m2 n2]=size(S)` where `S` is obtained by removing singletons from `A`. The singleton permutation places `A*P` in the form `[A11 A12 ; 0 S]` where `A11` is upper triangular with diagonal entries all greater than `opts.tol`.

  The default is to use COLAMD if `m2<=2*n2`; otherwise try AMD. Let `f` be the flops for `chol((S*P)'*(S*P))` with the ordering `P` found by AMD. Then if `f/nnz(R) >= 500` and `nnz(R)/nnz(S) >= 5` then try METIS, and take the best ordering found (AMD or METIS); otherwise use AMD without trying METIS. If METIS is not installed then the default ordering is to use COLAMD if `m2<=2*n2` and to use AMD otherwise.

  The available orderings are:

  `'default'`: the default ordering.

  `'amd'`: use `amd(S'*S)`.

  `'colamd'`: use `colamd(S)`.

  `'metis'`: use `metis(S'*S)`, only if METIS is installed.

  `'best'`: try all three (AMD, COLAMD, METIS) and take the best.

  `'bestamd'`: try AMD and COLAMD and take the best.

  `'fixed'`: use `P=I`; this is the only option if `P` is not present in the output.

  `'natural'`: singleton removal only.

- `opts.Q`: a string describing how `Q` is to be returned. The default is `'discard'` if `Q` is not present in the output, or `'matrix'` otherwise. If `Q` is present and `opts.Q` is `'discard'`, then `Q=[]` is returned (thus `R=spqr(A*P)` is `[Q,R,P]=spqr(A)` where `spqr` finds `P` but `Q` is discarded instead). The usage `opts.Q='matrix'` returns `Q` as a sparse matrix where `A=Q*R` or `A*P=Q*R`. Using `opts.Q='Householder'` returns `Q` as a struct containing the Householder reflections applied to `A` to obtain `R`, resulting in a far sparser `Q` than the `'matrix'` option.

- `opts.permutation`: a string describing how `P` is to be returned. The default is `'matrix'`, so that `A*P=Q*R`. Using `'vector'` gives `A(:,P)=Q*R` instead.

- `opts.spumoni`: an integer `k` that acts just like `spparms('spumoni',k)`.

- `opts.min2norm`: used by `spqr_solve`; you can use `'basic'` (the default), or `'min2norm'`. Determines the kind of solution that `spqr_solve` computes for underdetermined systems. Has no effect for least-squares problems; ignored by `spqr` itself.

2.4 Examples on how to use the MATLAB interface

To solve a least-squares problem, or to find the basic solution to an underdetermined system, just use `x = spqr_solve(A,b)` in place of `x=A\b`. To compute the QR factorization, use `[Q,R]=spqr(A)` instead of `[Q,R]=qr(A)`. Better results can be obtained by discarding `Q` with the usage `R=spqr(A)` (in place of `R=qr(A)`), or by requesting `Q` in Householder form with `[Q,R]=spqr(A,opts)` where `opts.Q='Householder'`. The latter option is not available in

MATLAB. To use a fill-reducing ordering, simply use any of the syntaxes above with `P` as an output parameter.

The least-squares solution of an overdetermined system `A*x=b` with `m>n` (where `A` has rank `n`) can be found in one of at least seven ways (in increasing order of efficiency, in time and memory):

`x = pinv(full(A)) * b ;`	impossible for large `A`
`[Q,R] = spqr (A) ;` `x = R\(Q'*b) ;`	high fill-in in `R`, `Q` costly in matrix form
`[Q,R,P] = spqr (A) ;` `x = P*(R\(Q'*b)) ;`	low fill-in in `R`, `Q` costly in matrix form
`[Q,R,P] = spqr (A,struct('Q','Householder')) ;` `x = P*(R\spqr_qmult (Q,b,0)) ;`	low fill-in in `R`, `Q` in efficient Householder form
`[c,R,P] = spqr (A,b) ;` `x = P*(R\c) ;`	`Q` not kept, `P` a permutation matrix
`[c,R,p] = spqr (A,b,0) ;` `y = (R\c) ; x(p) = y`	`Q` not kept, `p` a permutation vector
`x = spqr_solve (A,b) ;`	less memory and better handling of rank-deficient matrices

The minimum-norm solution of an underdetermined system `A*x=b` with `m<n` can be found in one of five ways (in increasing order of efficiency, in time and memory):

`x = pinv(full(A)) * b ;`	impossible for large `A`
`[Q,R] = spqr (A') ;` `x = Q*(R'\b) ;`	high fill-in in `R`, `Q` costly in matrix form
`[Q,R,P] = spqr (A') ;` `x = Q*(R'\(P'*b)) ;`	low fill-in in `R`, `Q` costly in matrix form
`[Q,R,P] = spqr (A',struct('Q','Householder')) ;` `x = spqr_qmult (Q,R'\(P'*b),1) ;`	low fill-in in `R`, `Q` in efficient Householder form
`opts.solution = 'min2norm' ;` `x = spqr_solve (A,b,opts) ;`	as 4th option above, but faster, less memory, and better handling of rank-deficient matrices

Note that `spqr_solve` uses a fill-reducing ordering, by default. It can be disabled or modified using a non-default `opts` parameter (`opts.ordering`, specifically).

3 Using SuiteSparseQR in C and C++

SuiteSparseQR relies on CHOLMOD for its basic sparse matrix data structure, a compressed sparse column format. CHOLMOD provides interfaces to the AMD, COLAMD, and METIS ordering methods, supernodal symbolic Cholesky factorization (namely, `symbfact` in MATLAB), functions for converting between different data structures, and for basic operations such as transpose, matrix multiply, reading a matrix from a file, writing a matrix to a file, and many other functions.

3.1 Installing the C/C++ library

In Linux/MacOs, type `make` at the command line, in either the `SuiteSparse` directory (which compiles all of SuiteSparse) or in the `SuiteSparse/SPQR` directory (which just compiles

SuiteSparseQR and the libraries it requires). SuiteSparseQR will be compiled, and a set of simple demos will be run (including the one in the next section).

The use of `make` is optional. The top-level `SPQR/Makefile` is a simple wrapper that uses `cmake` to do the actual build. The `CMakeLists.txt` file can be imported into MS Visual Studio, for example.

If SuiteSparseQR is compiled with `-DNEXPERT`, the "expert" routines in `SuiteSparseQR_expert.cpp` are not compiled. The expert routines are included by default.

To fully test 100% of the lines of SuiteSparseQR, go to the `Tcov` directory and type `make`. This will work for Linux only.

To install the shared library into /usr/local/lib and /usr/local/include, do `make install`. To uninstall, do `make uninstall`. For more options, see the `SuiteSparse/README.txt` file.

3.2 C/C++ Example

The C++ interface is written using templates for handling both real and complex matrices. The simplest function computes the MATLAB equivalent of `x=A\b` and is almost as simple:

```
#include "SuiteSparseQR.hpp"
X = SuiteSparseQR <double> (A, B, cc) ;
```

The C version of this function is almost identical:

```
#include "SuiteSparseQR_C.h"
X = SuiteSparseQR_C_backslash_default (A, B, cc) ;
```

Below is a simple C++ program that illustrates the use of SuiteSparseQR (with 64-bit integer indices). The program reads in a least-squares problem from `stdin` in MatrixMarket format [4], solves it, and prints the norm of the residual and the estimated rank of `A`. The comments reflect the MATLAB equivalent statements. The C version of this program is identical except for the `#include` statement and call to SuiteSparseQR which are replaced with the C version of the statement above, and C-style comments.

```
#include "SuiteSparseQR.hpp"
int main (int argc, char **argv)
{
    cholmod_common Common, *cc ;
    cholmod_sparse *A ;
    cholmod_dense *X, *B, *Residual ;
    double rnorm, one [2] = {1,0}, minusone [2] = {-1,0} ;
    int mtype ;

    // start CHOLMOD
    cc = &Common ;
    cholmod_l_start (cc) ;

    // load A
    A = (cholmod_sparse *) cholmod_l_read_matrix (stdin, 1, &mtype, cc) ;

    // B = ones (size (A,1),1)
    B = cholmod_l_ones (A->nrow, 1, A->xtype, cc) ;
```

```
    // X = A\B
    X = SuiteSparseQR <double> (A, B, cc) ;

    // rnorm = norm (B-A*X)
    Residual = cholmod_l_copy_dense (B, cc) ;
    cholmod_l_sdmult (A, 0, minusone, one, X, Residual, cc) ;
    rnorm = cholmod_l_norm_dense (Residual, 2, cc) ;
    printf ("2-norm of residual: %8.1e\n", rnorm) ;
    printf ("rank %ld\n", cc->SPQR_istat [4]) ;

    // free everything and finish CHOLMOD
    cholmod_l_free_dense (&Residual, cc) ;
    cholmod_l_free_sparse (&A, cc) ;
    cholmod_l_free_dense (&X, cc) ;
    cholmod_l_free_dense (&B, cc) ;
    cholmod_l_finish (cc) ;
    return (0) ;
}
```

To use SuiteSparseQR with 32-bit integer indices in all of its matrices, simply replace all `cholmod_l_*` calls abovce to `cholmod_*`, and use this for SPQR:

```
    X = SuiteSparseQR <double,int32_t> (A, B, cc) ;
```

3.3 C++ Syntax

All features available to the MATLAB user are also available to both the C and C++ interfaces using a syntax that is not much more complicated than the MATLAB syntax. Additional features not available via the MATLAB interface include the ability to compute the symbolic and numeric factorizations separately (for multiple matrices with the same nonzero pattern but different numerical values). The following is a list of user-callable C++ functions and what they can do:

1. `SuiteSparseQR`: an overloaded function that provides functions equivalent to `spqr` and `spqr_solve` in the SuiteSparseQR MATLAB interface.

2. `SuiteSparseQR_factorize`: performs both the symbolic and numeric factorizations and returns a QR factorization object such that `A*P=Q*R`. It always exploits singletons.

3. `SuiteSparseQR_symbolic`: performs the symbolic factorization and returns a QR factorization object to be passed to `SuiteSparseQR_numeric`. It does not exploit singletons.

4. `SuiteSparseQR_numeric`: performs the numeric factorization on a QR factorization object, either one constructed by `SuiteSparseQR_symbolic`, or reusing one from a prior call to `SuiteSparseQR_numeric` for a matrix `A` with the same pattern as the first one, but with different numerical values.

5. `SuiteSparseQR_solve`: solves a linear system using the object returned by `SuiteSparseQR_factorize` or `SuiteSparseQR_numeric`, namely `x=R\b`, `x=P*R\b`, `x=R'\b`, or `x=R'\(P'*b)`.

6. `SuiteSparseQR_qmult`: provides the same function as `spqr_qmult` in the MATLAB interface, computing `Q'*x`, `Q*x`, `x*Q'`, or `x*Q`. It uses either a QR factorization in MATLAB-style sparse matrix format, or the QR factorization object returned by `SuiteSparseQR_factorize` or `SuiteSparseQR_numeric`.

7. `SuiteSparseQR_min2norm`: finds the minimum 2-norm solution to an underdetermined linear system.

8. `SuiteSparseQR_free`: frees the QR factorization object.

3.4 Details of the C/C++ Syntax

For further details of how to use the C/C++ syntax, please refer to the definitions and descriptions in the following files:

1. `SuiteSparse/SPQR/Include/SuiteSparseQR.hpp` describes each C++ function. Both `double` and `std::complex<double>` matrices are supported.

2. `SuiteSparse/SPQR/Include/SuiteSparseQR_definitions.h` describes definitions common to both C and C++ functions. For example, each of the ordering methods is given a `#define`'d name. The default is `ordering = SPQR_ORDERING_DEFAULT`, and the default tolerance is given by `tol = SPQR_DEFAULT_TOL`.

3. `SuiteSparse/SPQR/Include/SuiteSparseQR_C.h` describes the C-callable functions.

Version 4.0 of SuiteSparseQR adds a 32-bit version, where the indices of its sparse matrices are all `int32_t`, contributed by Raye Kimmerer. To use this version, add `int32_t` as the second template parameter to all C++ methods, and use the `cholmod_*` methods instead of `cholmod_l_*` to create and access its input/output matrices.

The C/C++ options corresponding to the MATLAB `opts` parameters and the contents of the optional `info` output of `spqr_solve` are described below. Let `cc` be the CHOLMOD `Common` object, containing parameter settings and statistics. All are of type `double`, except for `SPQR_istat` which is `int64_t`, `cc->memory_usage` which is `size_t`, and `cc->SPQR_nthreads` which is `int`. Parameters include:

`cc->SPQR_grain`	the same as `opts.grain` in the MATLAB interface
`cc->SPQR_small`	the same as `opts.small` in the MATLAB interface
`cc->SPQR_nthreads`	the same as `opts.nthreads` in the MATLAB interface

Other parameters, such as `opts.ordering` and `opts.tol`, are input parameters to the various C/C++ functions. Others such as `opts.solution='min2norm'` are separate functions in the C/C++ interface. Refer to the files listed above for details. Output statistics include:

cc->SPQR_flopcount_bound	an upper bound on the flop count
cc->SPQR_tol_used	the tolerance used (opts.tol)
cc->SPQR_istat [0]	upper bound on nnz(R)
cc->SPQR_istat [1]	upper bound on nnz(H)
cc->SPQR_istat [2]	number of frontal matrices
cc->SPQR_istat [3]	unused
cc->SPQR_istat [4]	estimate of the rank of A
cc->SPQR_istat [5]	number of column singletons
cc->SPQR_istat [6]	number of row singletons
cc->SPQR_istat [7]	ordering used
cc->memory_usage	memory used, in bytes

The upper bound on the flop count is found in the analysis phase, which ignores the numerical values of A (the same analysis phase operates on both real and complex matrices). Thus, if you are factorizing a complex matrix, multiply this statistic by 4.

4 GPU acceleration

As of version 2.0.0, SuiteSparseQR now includes GPU acceleration. It can exploit a single NVIDIA GPU, via CUDA. The packages SuiteSparse_GPURuntime and GPUQREngine are also required (they should appear in the SuiteSparse directory, along with SPQR).

At run time, you must also enable the GPU by setting Common->useGPU to true. Before calling any SuiteSparseQR function, you must poll the GPU to set the available memory. Below is a sample code that initializes CHOLMOD and then polls the GPU for use in SuiteSparseQR.

```
size_t total_mem, available_mem ;
cholmod_common *cc, Common ;
cc = &Common ;
cholmod_l_start (cc) ;
cc->useGPU = true ;
cholmod_l_gpu_memorysize (&total_mem, &available_mem, cc) ;
cc->gpuMemorySize = available_mem ;
if (cc->gpuMemorySize <= 1)
{
    printf ("no GPU available\n") ;
}

// Subsequent calls to SuiteSparseQR will use the GPU, if available
```

See Demo/qrdemo_gpu.cpp for an extended example, which can be compiled via make gpu in the Demo directory.

GPU acceleration is not yet available via the MATLAB mexFunction interface. We expect to include this in a future release.

For a detailed technical report on the GPU-accelerated algorithm, see qrgpu_paper.pdf in the Doc directory.

5 Requirements and Availability

SuiteSparseQR requires four prior Collected Algorithms of the ACM: CHOLMOD [5, 10] (version 1.7 or later), AMD [1, 2], and COLAMD [8, 9] for its ordering/analysis phase and for its basic sparse matrix data structure, and the BLAS [11] for dense matrix computations on its frontal matrices; also required is LAPACK [3] for its Householder reflections. An efficient implementation of the BLAS is strongly recommended, either vendor-provided (such as the Intel MKL, the AMD ACML, or the Sun Performance Library) or other high-performance BLAS such as those of [13].

The use of Intel's Threading Building Blocks is optional [15], but without it, only parallelism within the BLAS can be exploited (if available). SuiteSparseQR can optionally use METIS 4.0.1 [14] and two constrained minimum degree ordering algorithms, CCOLAMD and CAMD [5], for its fill-reducing ordering options. SuiteSparseQR can be compiled without these ordering methods.

In addition to appearing as Collected Algorithm 8xx of the ACM, SuiteSparseQR is available at `http://www.suitesparse.com`. See SPQR/Doc/License.txt for the license. Alternative licenses are also available; contact the author for details.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, 1996.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):381–388, 2004.

[3] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 3rd edition, 1999.

[4] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. Chapman & Hall, London, 1997. (`http://math.nist.gov/MatrixMarket`).

[5] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Software*, 35(3), 2009.

[6] T. A. Davis. Algorithm 8xx: SuiteSparseQR, a multifrontal multithreaded sparse qr factorization package. *ACM Trans. Math. Software*, 2008. under submission.

[7] T. A. Davis. Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Software*, 2008. under submission.

[8] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):377–380, 2004.

[9] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Software*, 30(3):353–376, 2004.

[10] T. A. Davis and W. W. Hager. Dynamic supernodes in sparse Cholesky update/downdate and triangular solves. *ACM Trans. Math. Software*, 35(4), 2009.

[11] J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.

[12] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.

[13] K. Goto and R. van de Geijn. High performance implementation of the level-3 BLAS. *ACM Trans. Math. Software*, 35(1):4, July 2008. Article 4, 14 pages.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.

[15] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Sebastopol, CA, 2007.