LIBRSB: Universal Sparse BLAS Library

A highly interoperable Library for Sparse Basic Linear Algebra Subroutines for Multicore CPUs

Michele MARTONE

(Leibniz Supercomputing Centre, Garching bei München, Germany)

HPC, Big Data, and Data Science devroom at FOSDEM
Bruxelles, 05.02.2023

Systems of Linear Equations

$$\begin{cases} a_1 x + b_1 y &= c_1 \\ a_2 x + b_2 y &= c_2 \end{cases}$$

Matrix Representation

$$\begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

Exact Solutions

$$x = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = \frac{c_1 b_2 - b_1 c_2}{a_1 b_2 - b_1 a_2}$$

$$y = \frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} = \frac{a_1 c_2 - c_1 a_2}{a_1 b_2 - b_1 a_2}$$

Larger systems = ⚠

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \qquad\qquad\qquad &\vdots \quad \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

Larger systems = ⚠

$$
\begin{array}{rcl}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = & b_2 \\
\vdots & \vdots & \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = & b_n
\end{array}
$$

Complications

Larger systems = ⚠

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \qquad\qquad\quad & \vdots \quad \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

Complications

- numerical stability

Larger systems = ⚠

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad\qquad & \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

Complications

- numerical stability
- time to solution

What if...

- rows were several thousands

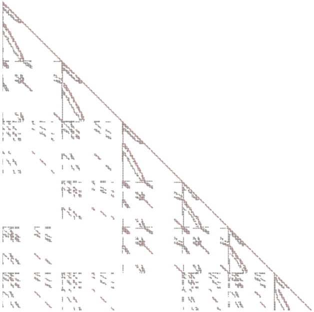
?

What if...

- rows were several thousands
- overwhelmingly populated by *zeros*

?

Is this ...*sparsity*?



12k rows, 300k non-zero elements, symmetric (here only lower triangle)

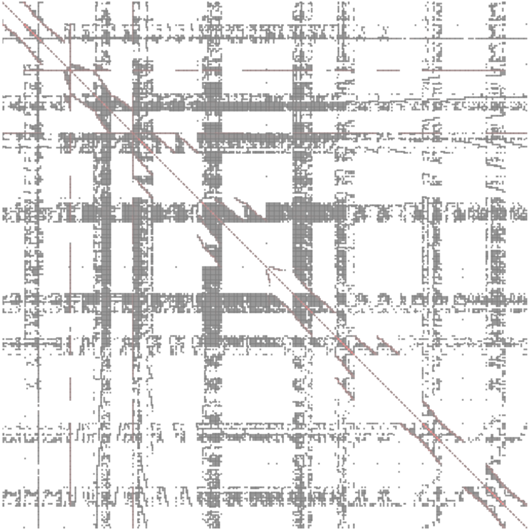
Occupation is ca. 0.4% (triangle only: 0.2%).
Matrix from `https://www.cise.ufl.edu/research/sparse/matrices/Cote/vibrobox`

Sparse Matrices

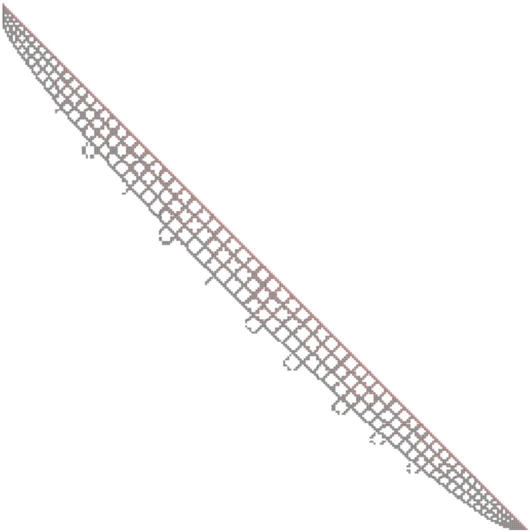
As attributed to James H. Wilkinson:

"any matrix with enough zeros that it pays to take advantage of them";

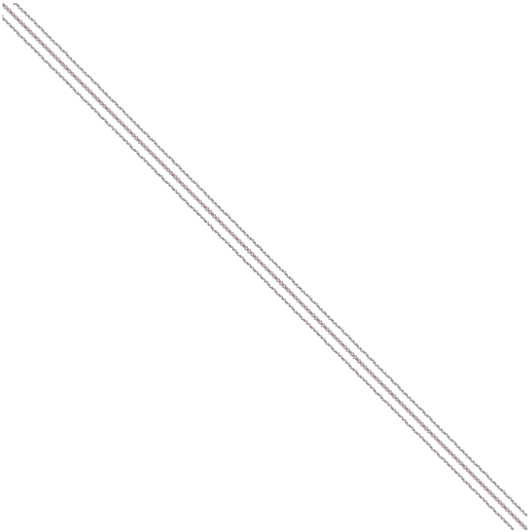
(see Tim Davis' letter http://www.netlib.org/na-digest-html/07/v07n06.html#2)



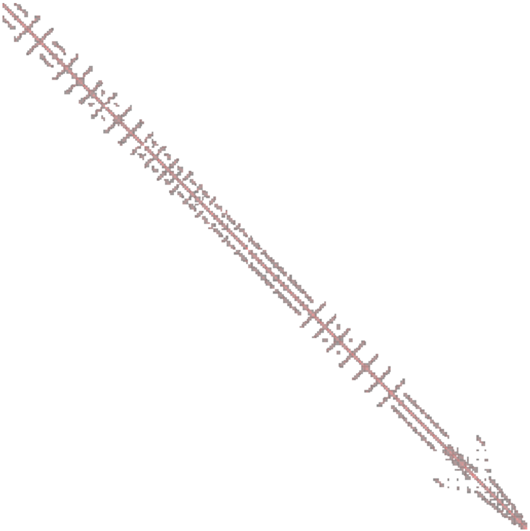
ASIC_320k



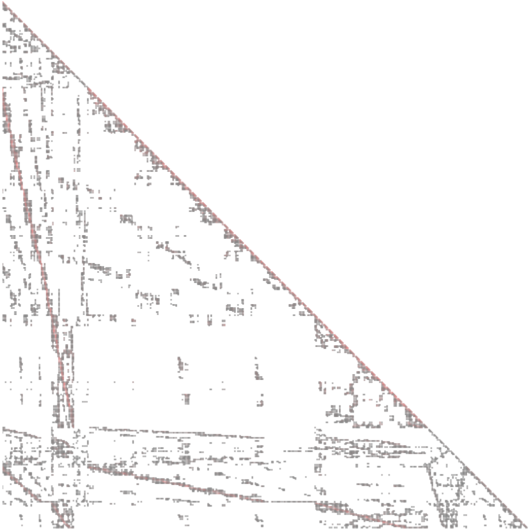
Ga41As41H72



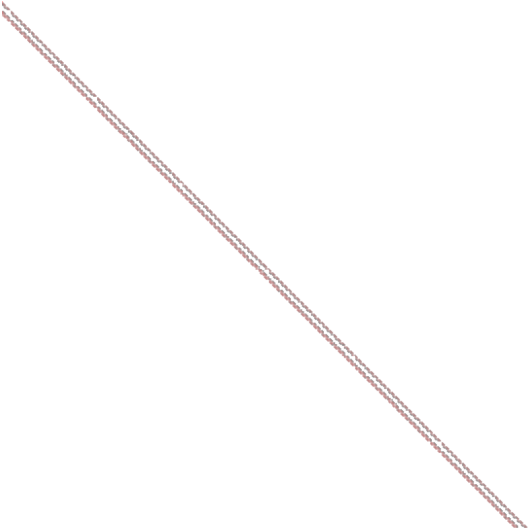
atmosmodl



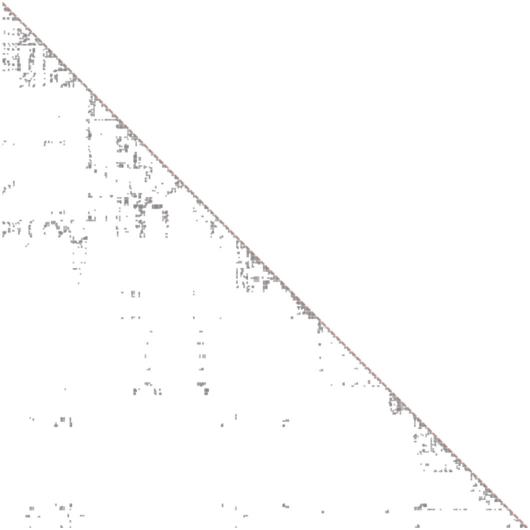
coater2



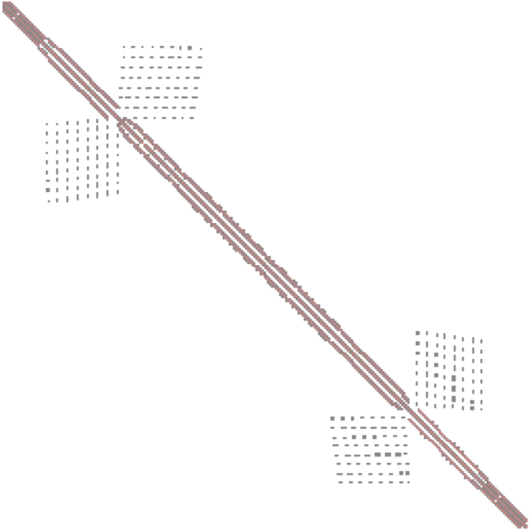
crankseg_1



crystk03



ct20stif



ex11

...Probably!

"Matrix sparsity"...

...depends on the computing techniques we choose.

Sparse Systems and Iterative Methods

```
function [x, flag, relres, iter, resvec] =
        cgs (A, b, tol, maxit, M1, M2, x0)
  ...
    elseif (isnumeric (A) && issquare (A))
      Ax = @(x) A * x;
    elseif (isa (A, "function_handle"))
      Ax = @(x) feval (A, x);
    ...
    for iter = 1:maxit
      ...
      q = Ax (p);
      alpha = ro / (p' * q);
      x = x + alpha * p;
      ...
```

Figure: GNU Octave's Conjugate Gradient Squared

Iterative Methods' frequent Bottlenecks

- SpMM – Sparse Multiply by dense Matrix: $A \cdot B$
- SpSM – Sparse (triangular) Solve by dense Matrix: $T^{-1} \cdot B$

Iterative Methods' frequent Bottlenecks

- SpMM – Sparse Multiply by dense Matrix: $A \cdot B$
- SpSM – Sparse (triangular) Solve by dense Matrix: $T^{-1} \cdot B$

If operand matrix is wide 1:

- SpMV – Sparse Multiply by dense Vector: $A \cdot x$
- SpSV – Sparse (triangular) Solve by dense Vector: $T^{-1} \cdot x$

SpMM can be defined as...

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

B, C:

SpMM can be defined as...

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian

B, C:

SpMM can be defined as...

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian
- rectangular, lower or upper triangular

B, C:

SpMM can be defined as...

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian
- rectangular, lower or upper triangular
- unit- or explicit- diagonal

B, C:

SpMM can be defined as...

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian
- rectangular, lower or upper triangular
- unit- or explicit- diagonal
- the four BLAS numerical types

B, C:

SpMM can be defined as...

$$\underbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op\left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian
- rectangular, lower or upper triangular
- unit- or explicit- diagonal
- the four BLAS numerical types
- optional: non-BLAS types, 64-bit indices

B,C:

SpMM can be defined as...

$$\underbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}_{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian
- rectangular, lower or upper triangular
- unit- or explicit- diagonal
- the four BLAS numerical types
- optional: non-BLAS types, 64-bit indices

B, C:

- by-rows or by-columns representation, custom leading dimension

SpMM can be defined as...

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{updated\ \ dense\ \ C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{dense\ \ C} + \alpha op \left(\overbrace{\begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}}^{sparse\ \ A} \right) \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{dense\ \ B}$$

$op(A)$:

- A, A^T, A'

A:

- general, symmetric, or hermitian
- rectangular, lower or upper triangular
- unit- or explicit- diagonal
- the four BLAS numerical types
- optional: non-BLAS types, 64-bit indices

B,C:

- by-rows or by-columns representation, custom leading dimension
- unit or non-unit stride

A secondary target problem: SpSM

$$\overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{\textit{updated dense } C} \leftarrow \beta \overbrace{\begin{bmatrix} c_{11} & \dots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nm} \end{bmatrix}}^{\textit{dense } C} + \alpha \overbrace{\begin{bmatrix} t_{11} & \dots & t_{1n} \\ \vdots & \ddots & \vdots \\ t_{n1} & \dots & t_{nn} \end{bmatrix}^{-1}}^{\textit{sparse } \textbf{triangular } T} \overbrace{\begin{bmatrix} b_{11} & \dots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nm} \end{bmatrix}}^{\textit{dense } B}$$

Options mostly as in SpMM.

Sparse BLAS: an API for iterative methods

- *Sparse BLAS = Sparse Basic Linear Algebra Subroutines*
- consolidated in 2001 by the *BLAS Technical Forum* (`http://www.netlib.org/blas/blast-forum/`)
- built around:
  - create and populate a matrix
  - SpMM, with all the options
  - SpSM, with all the options
  - destroy matrix
- API for C and Fortran (`blas_sparse.h` and `module blas_sparse`)

Merits and Reception

+ quite portable (on the CPU)
+ internal representation or parallelism are internal choices
+ one can imagine extensions...
- though no explicit extension specification
+ namesake and functionality adopted widely...
- but with slightly differing APIs
- no revision over the years (lost the GPU train...)

Merits and Reception

- + quite portable (on the CPU)
- + internal representation or parallelism are internal choices
- + one can imagine extensions...
- - though no explicit extension specification
- + namesake and functionality adopted widely...
- - but with slightly differing APIs
- - no revision over the years (lost the GPU train...)

Nevertheless

Still useful ☺

Sample SPARSE BLAS program (nothing LIBRSB-specific here)

example: spbusmv_pure.cpp

```
#include <blas_sparse.h>
int main(const int argc, char * const argv[]) {
  blas_sparse_matrix A = blas_invalid_handle;
  const int nnz = 4, nr = 3, nc = 3;
  const int    IA[] = { 0, 1, 2, 2 };
  const int    JA[] = { 0, 1, 0, 2 };
  double VA[] = { 11.0, 22.0, 13.0, 33.0  };
  double X[] = { 0.0, 0.0, 0.0 };
  const double B[] = { -1.0, -2.0, -2.0 };

  A = BLAS_duscr_begin(nr,nc);
  BLAS_ussp(A,blas_lower_symmetric);
  BLAS_duscr_insert_entries(A, nnz, VA, IA, JA);
  BLAS_duscr_end(A);
  BLAS_dusget_element(A, IA[0], JA[0], &VA[0]);
  BLAS_dusmv(blas_no_trans,-1,A,B,1,X,1);
  BLAS_usds(A);

  return 0;
}
```

RSB: Layout and Techniques for Sparse BLAS

- **R**ecursive **S**parse **B**locks
- does not exclude any Sparse BLAS option
- favours cache reuse with **cache blocking**
- **block-level**, coarse-grained threading
- block-level format can vary

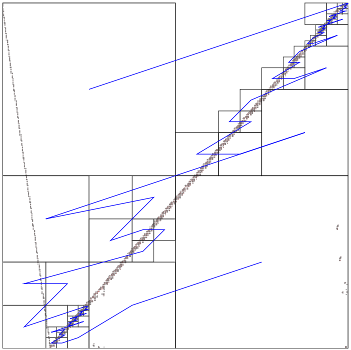
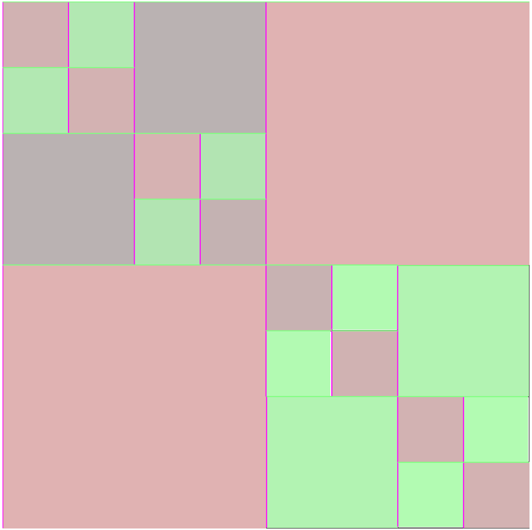
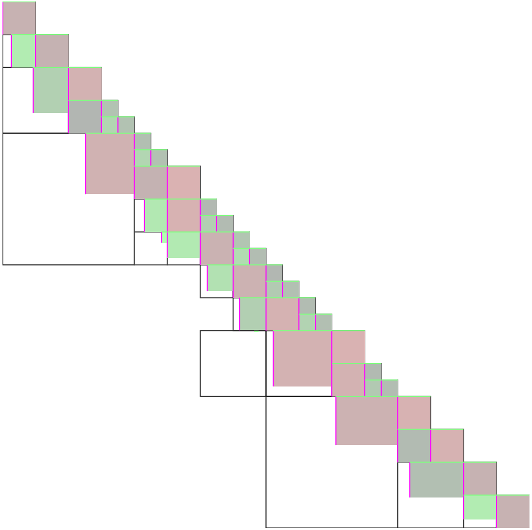


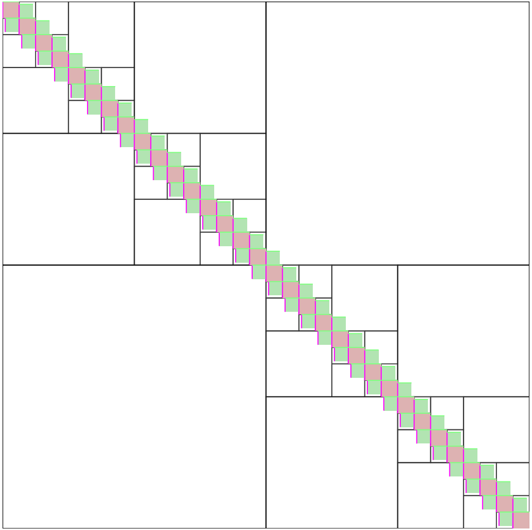
Figure: matrix `bayer02` as RSB



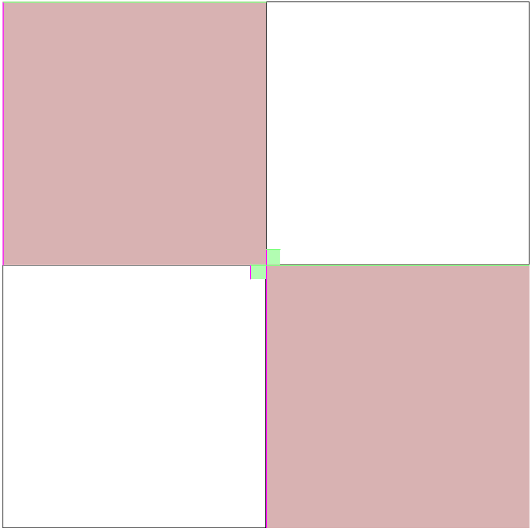
ASIC_320k as RSB



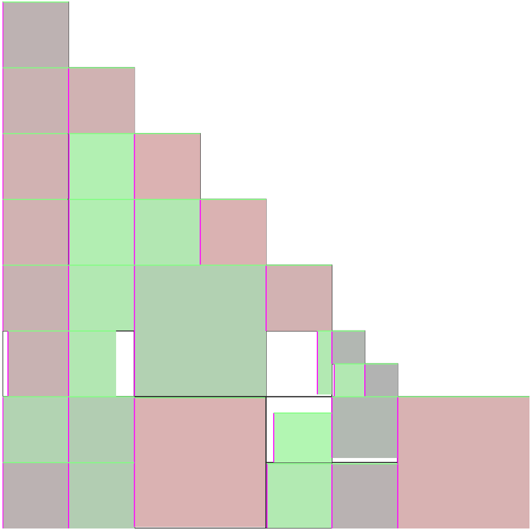
Ga41As41H72 as RSB



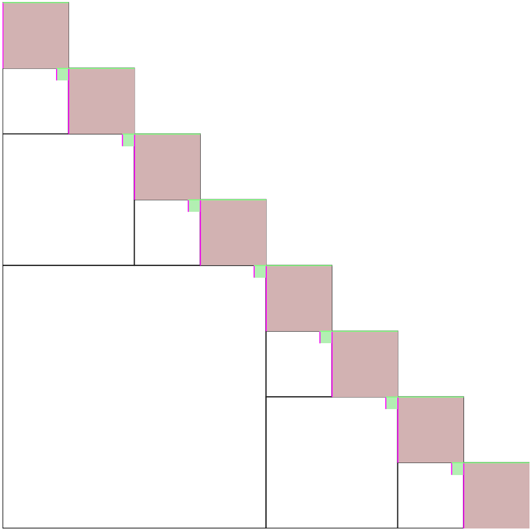
atmosmodl as RSB



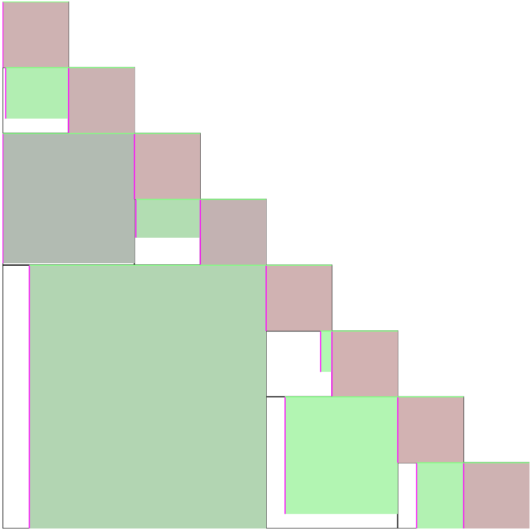
coater2 as RSB



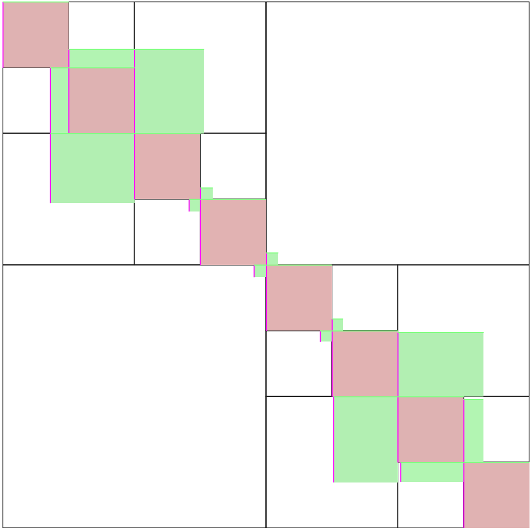
crankseg_1 as RSB



crystk03 as RSB



ct20stif as RSB



ex11 as RSB

LIBRSB: Shared Memory-parallel Sparse BLAS library around RSB

- *building blocks* for iterative solvers
  - interoperability
  - portability
  - compatibility

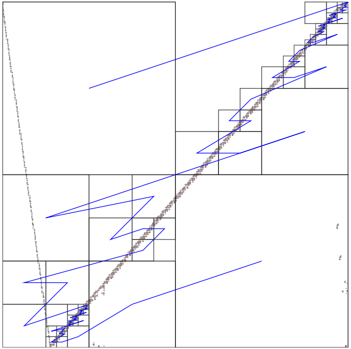


Figure: matrix `bayer02` rendered by LIBRSB

LIBRSB: Shared Memory-parallel Sparse BLAS library around RSB

- *building blocks* for iterative solvers
  - interoperability: **C, C++, Fortran, Octave, Python**
  - portability
  - compatibility

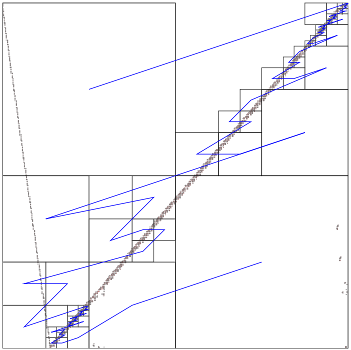


Figure: matrix `bayer02` rendered by LIBRSB

LIBRSB: Shared Memory-parallel Sparse BLAS library around RSB

- *building blocks* for iterative solvers
  - interoperability: **C, C++, Fortran, Octave, Python**
  - portability: **no intrinsics, POSIX, C99, OpenMP, C++...**
  - compatibility

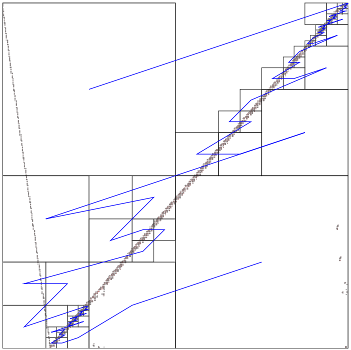


Figure: matrix `bayer02` rendered by LIBRSB

LIBRSB: Shared Memory-parallel Sparse BLAS library around RSB

- *building blocks* for iterative solvers
  - interoperability: **C, C++, Fortran, Octave, Python**
  - portability: **no intrinsics, POSIX, C99, OpenMP, C++...**
  - compatibility: **no fill-in, user-provided arrays,...**

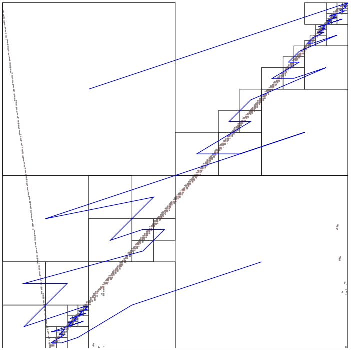


Figure: matrix `bayer02` rendered by LIBRSB

Sample SPARSE BLAS Program with LIBRSB

example: spbusmv.cpp

```
#include <rsb.h>
#include <blas_sparse.h>
int main(const int argc, char * const argv[]) {
  blas_sparse_matrix A = blas_invalid_handle;
  const int nnz = 4, nr = 3, nc = 3;
  const int    IA[] = { 0, 1, 2, 2 };
  const int    JA[] = { 0, 1, 0, 2 };
  double VA[] = { 11.0, 22.0, 13.0, 33.0  };
  double X[] = { 0.0, 0.0, 0.0 };
  const double B[] = { -1.0, -2.0, -2.0 };
  rsb_lib_init(RSB_NULL_INIT_OPTIONS);
  A = BLAS_duscr_begin(nr,nc);
  BLAS_ussp(A,blas_lower_symmetric);
  BLAS_duscr_insert_entries(A, nnz, VA, IA, JA);
  BLAS_duscr_end(A);
  BLAS_dusget_element(A, IA[0], JA[0], &VA[0]);
  BLAS_dusmv(blas_no_trans,-1,A,B,1,X,1);
  BLAS_usds(A);
  rsb_lib_exit(RSB_NULL_EXIT_OPTIONS);
  return 0;
}
```

Native LIBRSB APIs

Apart from `blas_sparse.h` and `module blas_sparse`...

- C/C++ in `rsb.h`
- C++ in `rsb.hpp`
- Fortran in `rsb.F90` (after `rsb.h`)

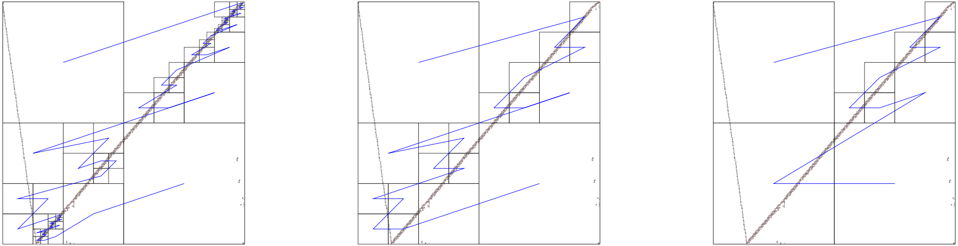
Native LIBRSB APIs

Apart from `blas_sparse.h` and `module blas_sparse`...

- C/C++ in `rsb.h`
- C++ in `rsb.hpp`
- Fortran in `rsb.F90` (after `rsb.h`)

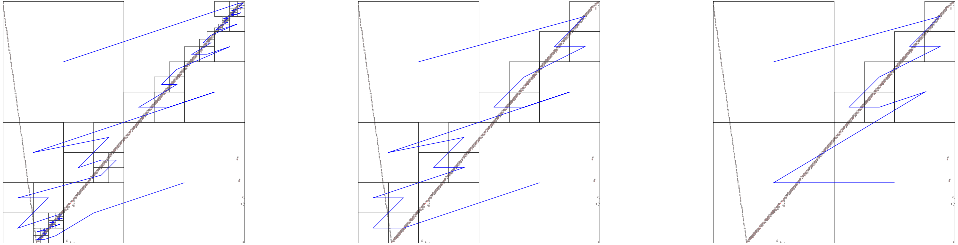
Why another API?

For extras, especially RSB-specific ones; see next slide.



Blocking granularity affects SpMM performance

Automated Empirical Optimization



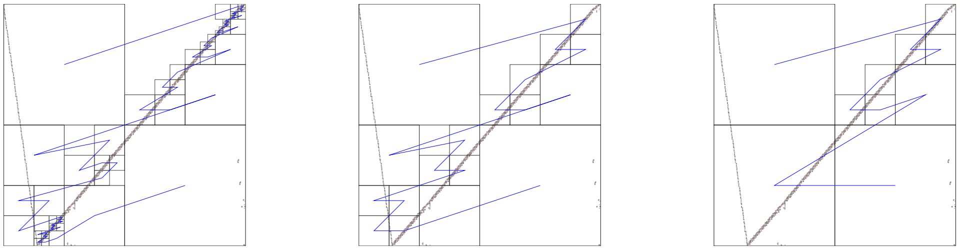
Blocking granularity affects SpMM performance

Automated search of *better* blockings

Essential to utilize LIBRSB at best

Trades off *search time* for savings in SpMM iterations.

Automated Empirical Optimization



Blocking granularity affects SpMM performance

Automated search of *better* blockings

Essential to utilize LIBRSB at best

Trades off *search time* for savings in SpMM iterations.

`rsb_tune_spmm()` in C, ...

C++ example

example: `nozero_span.cpp`

```
#include <array>
#include <vector>
#include <iostream>
#include <rsb.hpp>
int main()
{
  const rsb::RsbLib rsblib;
  const std::vector<rsb_coo_idx_t> IA {0,0,1,1}, JA {0,1,0,1};
  const std::array<double,4> VA {1,0,0,1};
  // square full matrix, no zeroes:
  rsb::RsbMatrix<double> mtx(IA,JA,VA,4,
     RSB_FLAG_DEFAULT_RSB_MATRIX_FLAGS | RSB_FLAG_DISCARD_ZEROS);
  std::cout << mtx.nnz();
}
```

```
2
```

Example: sparsersb in GNU Octave

```
octave:1> R=(rand(3)>.6)
R =

   0   0   0
   0   0   0
   1   0   1

octave:2> A_octave=sparse(R)
A_octave =

Compressed Column Sparse (rows = 3, cols = 3, nnz = 2 [22%])

  (3, 1) ->  1
  (3, 3) ->  1

octave:3> A_librsb=sparsersb(R)
A_librsb =

Recursive Sparse Blocks  (rows = 3, cols = 3, nnz = 2 [22%])

  (3, 1) -> 1
  (3, 3) -> 1
```

https://sourceforge.net/p/octave/sparsersb/ci/default/tree/src/sparsersb.cc

Example: LIBRSB + CYTHON = PYRSB

```
import numpy
import scipy
from scipy.sparse import csr_matrix
from rsb          import rsb_matrix

V = [11.,12.,22.]
I = [  0,  0,  1]
J = [  0,  1,  1]

c = csr_matrix((V,(I,J)))
y = y + c * x;

a = rsb_matrix((V,(I,J)))
y = y + a * x;
```

https://github.com/michelemartone/pyrsb

Default licensing of LIBRSB

Lesser GPLv3, aka LGPLv3

- unmodified, can be combined with proprietary software
- modified, must be released as LGPLv3

Availability

Sources:

http://librsb.sf.net/

Packaged:

- Linux distros: Debian, Ubuntu& derivatives, openSUSE, NixOS, AUR, ...
- Misc: FreeBSD, Cygwin
- Spack, Easybuild

Third party accessors

- Julia, Rust