CSPARSE

A MATLAB® Class for Compiled Sparse Computations

João P. Hespanha

September 8, 2014

Abstract

This class enables very fast repeated computations of (potentially sparse) tensors (i.e., multi-dimensional arrays). The speed gains are enabled by (i) detecting at compile time the (structural) sparsity patterns of the final results and also of all intermediate variables, (ii) determining at compile time all the memory indexing to access the sparse arrays (iii) performing at initialization time all the memory allocation needed for the intermediate results, (iv) reusing as much as possible previously used computations, (v) performing most of the run-time computations by optimized C functions.

Contents

1 Phylosophy

We start by introducing the key concepts behind `CSparse` that differentiate this tool from "standard" MATLAB®-to-C compilers.

1.1 When to use `CSparse`?

The `CSparse` class is used when one needs to perform computations involving tensors (i.e., multi-dimensional arrays) and results in very significant gains in computation speed in any of the following scenarios:

1. The final results or some of the intermediate computations involve tensors that are structurally sparse (i.e., with many entries that are equal to zero); e.g., `C=A.*diag(b)` (where `b` is a vector).

   An entry of a tensor is said to be *structurally zero* if once can (symbolically) guarantee that it is always zero; e.g., as in the non-diagonal entries of `diag(b)`. *Structural sparsity*, refers to the pattern of entries of a tensor that are structurally zero.

2. One needs to perform several tensor computations that share intermediate results that can be reused; e.g. `C=A*B+C;D=A*B-C`.

3. One needs to perform the same computation multiple times and one can reuse intermediate results from one computation to the next; e.g., `for i=1:5; C=C+A*B; end`.

4. One needs to perform the same computation multiple times and therefore one can reuse previously allocated memory from one computation to the next; e.g., `for i=1:5; C=C+rand(N); end`.

5. One needs to perform the computation using C code that does not use specialized libraries.

While illustrative, the examples under items 1–3 above are sufficiently simple that it would be straighforward to optimize the MATLAB® code to take advantage of the special structures of the expressions. The goal of the `CSparse` class is to do this automatically for much more complicated expression.

1.2 `CSparse` compile-time optimization

The CS class takes as inputs a collection of `TensCalcTools` symbolic tensor-valued expressions (STVEs) and generates C code to evaluate these expressions. The code generation process performs two important operations at *compile time* to enable very fast *run-time* evaluation of the STVEs:

1. It detects the *structural sparsity patterns* of the STVEs to be computed, as well as the sparsity patterns of all intermediate expressions needed to evaluate the STVEs. This permits the discovery (at compile time) of the total memory that needs to be allocated to perform all the computations and also to precompute all the indexing needed to access the sparse multi-dimensional arrays in memory.

2. It breaks all the computations into elementary operations and builds a *dependency tree* for these operations to determine all the computations that can be resused from the computation of one STVE to another, within a single STVE, or across successive computations of the same set of STVEs.[1]

[1]The dependency tree could also be used to distribute computation among several processors/threads, but currently this is not done and a single thread is used.

2 CSparse declarations

The C code generated uses a single continuous one-dimensional array to store all the (nonzero) entries of all the `TensCalcTools` tensors with input data, all the intermediate computations, and the output data. We call this array the *scratchboard*.

The following `CSparse` functions are used to determine which STVEs one wants to specify which STVEs one wants to compute, which `TensCalcTools` variables should be regarded as input parameters. These declarations implicitely allocate blocks of the scratchbook to all the `TensCalcTools` tensors needed to perform the computations.

`csparse`

```
code=csparse();
```

This function returns an empty `CSparse` object that will eventually be compiled into run-time code (after adding STVEs to it). The scratchbook associated with this object is initially empty.

`declareSet`

```
declareSet(code,STVE,'funname');
```

This function adds the given `TensCalcTools` STVE to the `CSparse` object `code`, specifying that the STVE should be regarded as an input variable to be provided externally to the run-time `CSparse` code. `STVE` is taken to be a *structurally full* tensor in the sense that all its entries are assumed to be nonzero.

The function `declareSet` implicitly requests the creation of a C function `void funname(double *inputData)` that takes a pointer `inputData` to a (non-sparse) tensor with input data and copies this data to an appropriate location in the scratchboard for subsequent computations. This C function keeps track in run-time of which entries of the scratchboard have been modified to determine which (dependent) entries of the scratchboard need to be subsequently recomputed.

Typically, `STVE` is a TensCalcTools() `Tvariables` or some direct indexing of a `Tvariables`, as in

```
% MATLAB declarations
Tvariable x [5]
Tvariable y [5]
z = 2*(x+y)
declareSet(code,x,'setX')
declareSet(code,y(1:3),'setY13')
declareSet(code,y(4:5),'setY45')
```

which could be used to eventually create code that compute $2 * (x + y)$ using the collowing C code

```
// C code
double x[5],y13[3],y45[2];
// ... code to compute x, y13, y45 goes here ...
setX(&x);
setY(&y13);
setY(&y45);
// ... code to retrieve z from the scratchbook goes here (see declareGet())
```

STVEs that are more general TensCalcTools expressions may lead to "unexpected" results: e.g., with

```
% MATLAB declarations
Tvariable x [5]
Tvariable y [5]
z = 2*(x+y)
declareSet(x+y,'setX_plus_Y')
```

the C function `setX_plus_Y()` allows one to set directly the intermediate expression $x + y$, which is then doubled to obtain z (without ever adding x and y). However, in the "apparently similar" code

```
% MATLAB declarations
Tvariable x [5]
Tvariable y [5]
z = 2*x+2*y
declareSet(x+y,'setX_plus_Y')
```

the C function `setX_plus_Y()` allows one to set values in the scratchbook for the expression $x + y$, but the computation of z does not explicitely use $x + y$ so by setting the value of $x + y$ this does not affect the value of z. This arises because no attempt is mode within TensCalcTools or CSparse to identify $2 * (x + y)$ with $2 * x + 2 * y$. In fact, these toolbox do not recognize many algebraic rule (including the distributivity rule, etc.). However, they do recognize a few rules (including the roles of the multiplicative and additive identities and some forms of commutativity and associativity). Since the user cannot be sure of what types of symbolic manipulations will be done internally, one may get "unpredictable" behavior when `STVE` in the `declareSet` is not a `Tvariables` or some direct indexing of a `Tvariables`

declareGet

```
declareGet(code,STVE,'funname');
```

This function adds the given TensCalcTools STVE to the CSparse object `code`, specifying that the STVE should be regarded as an output variable to be computed by the run-time CSparse code. `STVE` must be a *structurally full*[2] tensor in the sense that all its entries are assumed to be nonzero. If that is not the case, one can always use

```
declareGet(code,full(STVE),'funname');
```

where the `full` operator "fills-in" any structurally nonzero entries.

The function `declareGet` implicitly requests the creation of a C function `void funname(double *outputBuffer)` that that performs any required recomputations needed to obtain the value of `STVE` and copies the corresponding locations of the scratchbook to `outputBuffer`. This C function keeps track of which entries of the scratchbook have been modified (by previous calls to C functions generated by `declareGet` and `declareCopy`) to make sure that a minimum ammount of scratchbook entries are recomputed.

declareCopy

```
declareCopy(code,STVEdest,STVEsrc,'funname');
```

[2] The current implementation of CSparse actually allows `declareGet` to be called with a sparse STVE for 2-dimensional tensors. However, this has not been tested.

This function adds the given TensCalcTools STVEs to the CSparse object code. The given STVEs may be full or sparse, but in the latter case they need to have the same exact structural sparsity pattern.

The function declareCopy implicitly requests the creation of a C function void funname() that copies the value of STVEsrc in the scratcbook to the location of STVEdest in the scratchbook. Like the C functions created by declareGet, this C function keeps track of which entries of the scratchbook have been modified (by previous calls to C functions generated by declareGet and declareCopy) to make sure that a minimum ammount of scratchbook entries are recomputed. In addition, like the C functions created by declareSet, this C function keeps track in run-time of which entries of the scratchboard have been modified to determine which (dependent) entries of the scratchboard need to be subsequently recomputed.

As with declareSet, care must be exercise when the destination STVE STVEdest are not Tvariable.

Unlike declareSet and declareGet, declareCopy may operate on groups of tensors, i.e., STVEsrc and STVEdest may be cell arrays of STVEs. This is important because the copy is carried out atomically without regard to dependencies betweem the tensors in the group. To understand this consider the two alternative MATLAB® declarations and the corresponding C code that calls the copy functions so generated:

```
% MATLAB declarations
declareCopy(code,{A2,B2},{A1,B1},'copyA1B1toA2B2');
\\ C code
copyA1B1toA2B2();
```

and

```
% MATLAB declarations
declareCopy(code,A2,A1,'copyA1toA2');
declareCopy(code,B2,B1,'copyB1toB2');
\\ C code
copyA1toB1();
copyA2toB2();
```

In the first option copyA1B1toA2B2() starts by (re)computing A1,B1, then makes the copy, and finally marks A2,B2 as having been modified (which eventually may trigger other computations). In the second option copyA1toB1();copyA2toB2(); starts by (re)computing A1, then makes the copy of the value of A1 to A2, then marks A2 as modified, and only after that re(computes) B2 — potentially taking into account the changes made to A2.

declareAlias

```
[SVTEout]=declareAlias(code,STVE,'name');
[SVTEout]=declareAlias(code,STVE,'name',atomic);
[SVTEout,subscripts]=declareAlias(code,STVE,'name');
```

This function adds the given TensCalcTools STVE to the CSparse object code without declaring it an input or output variable, but returns a TensCalcTools STVE Tvariable called name that is linked to the value of STVE in the scratchbook.

When called with more than 1 output argument, the sparsity structure of the tensor STVE is returned in subscripts, which is a matrix with one row per dimension of the tensor STVE and one column per structurally nonzero entry of STVE. Each column of subscripts contains the subscripts of a structurally nonzero entry of STVE.

Atomic operations When the boolean input parameter atomic is true, the C-code used to compute the top operator of the STVEs will not store its output tensor entries in the scratchbook. Instead, its output will be dynamically allocated/deallocated as a block. This has several consequences:

1. The operator's output is not stored statically in the scratchbook. Instead, each time it needs to be recomputed, memory is allocated to store its value. In case a previous version of the computation existed in memory, it is freed prior to allocating the memory for the new computation.

2. Atomic operators can be computed by algorithms for which the sparsity structure of the result cannot be determined at compile time, such as the LU factorization of a sparse matrix with pivoting adapted to the specific matrix.

3. Atomic operators are always recomputed as a whole, even if only a small subset of its operands have changed.

Declaring an operator as *atomic* will not affect the generation of MATLAB® code.

Working with aliases. Aside from learning at compile time the structural sparsity pattern of STVE (when called with multiple output arguments), the main use of declareAlias is to obtain a "simple" TensCalcTools Tvariable that represents a (potentially very complex) TensCalcTools expression, which reduces the overhead of subsequent TensCalcTools symbolic manipulations. To understand this, consider the following declaration

```
% MATLAB declarations
J=norm2(A*x-b);
g=gradient(J,x);
z=g(1:3)-g(4:6);
declareGet(code,z,'getZ')
```

which results in the following values of the TensCalcTools STVEs

```
g=mytprod(A,[1,-1],x,[-1])-b,[-1],A,[-1, 1]);    % same as A'*(A*x-b)
z= subsref(mytprod(2,[],mytprod(A,[1,-1],x,[-1])-b,[-1],A,[-1, 1]),...
           struct('type','()','subs',{{(1:3)}}))...
  -subsref(mytprod(2,[],mytprod(A,[1,-1],x,[-1])-b,[-1],A,[-1, 1]),...
           struct('type','()','subs',{{(4:6)}}))
```

where one can see the gradient appearing twice in z. In spite of this, TensCalcTools is "smart enough" to detect the duplication and generates code tha does not compute the gradient twice when it evaluates z. Nevertheless, TensCalcTools STVEs tend to grow rapidly and their processing results can result in large overhead — at compile time, not at run time!

Unwieldy TensCalcTools STVEs can be avoided in declareAlias, as in

```
% MATLAB declarations
J=norm2(A*x-b);
g=gradient(J,x);
g=declareAlias(code,g,'z');
z=g(1:3)-g(4:6);
declareGet(code,z,'getZ')
```

which results in the following values of the TensCalcTools STVEs

```
g=mytprod(A,[1,-1],x,[-1])-b,[-1],A,[-1, 1]);    % same as A'*(A*x-b)
z=g(1:3)-g(4:6);
```

with the understanding that CSparse internally knows that `g` is actually given by the expression in line 55 and will take that into account in the generation of any C code. However, as far as TensCalcTools goes `g` is a freshly created new variable, unrelated to `A`, `x`, or `b`.

Two items must be taken into account:

1. The link between the input STVE `STVE` and the output STVE `STVEout` is known to the CSparse object `code`, but the dependence of `STVEout` with respect to other variables that appear in `STVE` is unkown to TensCalcTools.

   This affects TensCalcTools's ability to perform some symbolic manipulations. E.g., `gradient(z,x)` will return `A'*A` when `z` is computed using line 46, but will return 0 when `z` is computed using line 57.

2. Space will be reserved in the scrapbook to hold the tensor `STVE` and all the intermediate tensors needed to compute it, regardless of whether or not this tensor is needed for the C functions generated by `declareSet` and `declareGet`, `declareCopy`.

declareSave

```
declareSave(code,STVE,'funname','filename_subscripts');
```

This function adds the given TensCalcTools STVE to the CSparse object `code` and writes the sparsity structure of the STVE to a (binary) file called `filename_subscripts`.

The function `declareSave` implicitely request the creation of a C function `void funname(char *filename_values)` that writes to a (binary) file called `filename_values` the values in the scrapbook corresponding to the given STVE. Opposite to the C function created by `declareGet`, the C function created by `declareSave` *does not* trigger any recomputations of values in the scrapbook; it simply write the current values in the scrapbook.

The structures of the files `filename_subscripts` and `filename_values` are described in Tables 1 and 2, respectively. These files can be read using the CSparse command

```
[subscripts,values]=loadCSparse(filename_subscripts,filename_values)
```

The filename can also be passed to the TensCalcTools function `lu` as a "typical" value for the matrix to be factorized. This "typical" value is used by CSparse to determine "optimal" pivoting, row/column permutations (and potentially scaling[3]).

declareCfunction

```
declareCfunction(code,'funfile.c','funname',inputs,outputs,defines);
```

[3] Not yet implemented.

Table 1: Structure of the binary file `filename_subscripts`.

name	length	description
magic	1 x sizeof(int64_t)	random number that is common to filename_subscripts and filename_values that can be used to make sure the the two files are compatible
nDim	1 x sizeof(int32_t)	number of dimensions of the tensor
osize	nDim x sizeof(int64_t)	size of the tensor in each dimension
subscripts	nDim x nnz x sizeof(int64_t)	matrix with 0-based subscripts of the nonzero dimensions (one nonzero entry after another)

Table 2: Structure of the binary files `filename_values`.

name	length	description
magic	1 x sizeof(int64_t)	random number that is common to filename_subscripts and filename_values that can be used to make sure the the two files are compatible
values	nnz x sizeof(double)	values of the nonzero entries in the order the subscripts appear in the file filename_subscripts.

This functions adds to the CSparse object `code` a C function called `void funname()` found in the file `funfile.c`. This function does not operate directly on the scrapbook, but will typically call C functions created through `declareSet`, `declareGet`, `declareCopy`, and `declareSave`.

The function declared is of type `void funname()` with its input parameters defined by the structure arrays `inputs` and `outputs`, respectively, according to the following templates:

```
inputs(1).name = {string with the name of the variable}
inputs(1).type = {string describing the matlab input type as in
                  (uint8|uint16|uint32|uint64|int8|int16|int32|int64|float|double)}
inputs(1).size = {numeric array with the size of matrix}
inputs(2).type = ...
   ...
outputs(1).name = {string with the name of the variable}
outputs(1).type = {string describing the matlab input type as in
                  (uint8|uint16|uint32|uint64|int8|int16|int32|int64|float|double)}
outputs(1).size = {numeric array with the size of matrix}
outputs(2).type = ...
```

All inputs and outputs are passed by reference, with all the inputs first, followed by the outputs. The structure `defines` specifies a set of `#define` pre-processor directives that should precede the C function definition and can be used to pass (hardcoded) parameters to the C function, according to the following templates:

```
defines.name1 = {string or scalar}
defines.name2 = {string or scalar}
   ...
```

saveVectorized

```
saveVectorized(code,filename);
```

This function saves the `CSparse` object as a computational graph whose nodes are functions that operate on STVEs, using the format described in `computationalGraphs.pdf`. The filenames for the different files are constructed from `filename`, which should not have an extension.

`compile2C`

```
compile2C(code);
```

3 Run-time Considerations

The existence of the scratchboard is transparent to the user, who does not need to worry about where in the scratchboard values will be stored or how they are addressed. Because of this, the user does not need to pass any information about scratchboard locations to the functions `declareSet`, `declareGet`, and `declareCopy`. Obviously, `CSparse` object internally keep track of where everything is in the scratchboard.

From the C-code prespective, the scratchboard is an one-dimensional array of floating point variables (typically double for matlab compatility) that is accessed by all the run-time C functions as a global variable. The whole array can either be declared as a global variable or it can be allocated in run time using `malloc`. In the latter case, two C functions named `initializer()` and `finalizer()` are generated to allocate the scratchbook and deallocate it, respectively; and any calls to the C functions generated through `declareSet`, `declareGet`, and `declareCopy` must be made between calls to `initializer()` and `finalizer()`. Aside from the scratchbook, a few more global variables are used to keep track of scratchbook dependencies.

4 Quick Start

The following code demonstrates the use CSparse.

4.1 Standalone C code

We first show how to use CSparse to create a standalone C program that solves a simple optimization using gradient descent.

We start by defining the dimensions of the variables an constants to be used later:

```
N=10000;
n=800;
b=rand(N,1);
```

We are now ready to define the TensCalcTools STVEs that we want to compile. To do this one uses Tvariable to define symbolic variables, which one can then use in several matlab functions that have been redefined to accept such variables. The new function gradient permits symbolic differentiation. At this time no computations are performed.

```
Tvariable A [N,n];
Tvariable x n;
y=A*x-b;
J=norm2(y);
grad=gradient(J,x);
ngrad2=norm2(grad);
xx=x-.1/(N*n)*grad;
```

The next step is to define the computations that we want to compile. The class csparse is used for this purpose, with the methods declareSet used to declare input variables and expressions that we want to compile, declareGet to declare the output variables that we want to compute, and declareCopy to declare assignments between variables that we want to do in run-time.

```
code=csparse();
declareSet(code,A,'setA');           % ask for function to set value of A
declareSet(code,x,'setX');           % ask for function to set value of x
declareGet(code,J,'getJ');           % ask for function to get value of J
declareGet(code,x,'getX');           % ask for function to set value of xx
declareGet(code,ngrad2,'getNgrad2'); % ask for function to get value of getNgrad2
declareCopy(code,x,xx,'copyXx2X');   % ask for a function to copy value of xx to x
```

The method compile2C now generates the appropriate C code:

```
!rm -f tmpC_docStandAlone.c          % erase previous version since compile2C appends
compile2C(code,'tmpC_docStandAlone.c');
```

Finally, the computations are ready to be performed within a C program, e.g., using the following standalone code:

```
#include <stdlib.h> // needed for rand()
#include "tmpC_docStandAlone.c"

#define N 100
#define n 8
```

```
int main()
{
  double A[N*n],x[n],J,ngrad2,gamma=.1;
  int i;

  for (i=0;i<N*n;i++) A[i]=(double)rand()/(double)RAND_MAX;
  for (i=0;i<n;i++) x[i]=(double)rand()/(double)RAND_MAX;

  printf("Before:\n");
  getNgrad2(&ngrad2);
  getJ(&J);
  printf("  ngrad2 = %10lf, J = %10lf\n",ngrad2,J);

  for (i=0;;i++) {
    getNgrad2(&ngrad2);
    if (ngrad2<1e-3) break;
    copyXx2X();
  }

  printf("After %d iterations:\n",i);
  getX(x);
  for (i=0;i<n;i++)
    printf("  x[%3d] = %10lf\n",i,x[i]);
  getNgrad2(&ngrad2);
  getJ(&J);
  printf("  ngrad2 = %10lf, J = %10lf\n",ngrad2,J);
}
```

4.2 C-code called from MATLAB®

We now show how to use CSparse to create a cmex MATLAB® function that solves the same simple oprimization using gradient descent. The cmex function now takes the input data from MATLAB® variable and returns the result also to MATLAB® variables. To accomplish this we use the cmexTools toolbox.

The following cmexTools template does the trick:

```
#ifdef createGateway

function tmpC_docCmex        % name of the cmex (gateway) function
Cfunction docCmex_raw        % function called by the gateway for the computations
include docCmex.c            % include this function before the gateway

inputs                       % inputs to the cmex function
      double A[N,n]
      double x0[n]
outputs                      % outputs of the cmex function
      double J[1]
      double x[n]

preprocess(N,n,b)            % matlab code executed before creating the gateway function
     Tvariable A [N,n];
     Tvariable x n;

     y=A*x-b;
     J=norm2(y);
```

```
    grad=gradient(J,x);
    ngrad2=norm2(grad);
    xx=x-.1/(N*n)*grad;

    code=csparse();
    declareSet(code,A,'setA');           % ask for function to set value of A
    declareSet(code,x,'setX');           % ask for function to set value of x
    declareGet(code,J,'getJ');           % ask for function to get value of J
    declareGet(code,x,'getX');           % ask for function to set value of xx
    declareGet(code,ngrad2,'getNgrad2'); % ask for function to get value of getNgrad2
    declareCopy(code,x,xx,'copyXx2X');   % ask for a function to copy value of xx to x

    compile2C(code,'tmpC_docCmex.c');    % to be appended to the gateway function
#endif

void docCmex_raw(/* inputs */
                 double *A,
                 double *x0,
                 /* outputs */
                 double *J,
                 double *x,
                 /* sizes */
                 mwSize N,
                 mwSize n)
{
  double ngrad2;
  int i;

  setA(A);
  setX(x0);

  printf("Before:\n");
  for (i=0;i<n;i++)
    printf("   x[%3d] = %10lf\n",i,x[i]);
  getNgrad2(&ngrad2);
  getJ(J);
  printf("   ngrad2 = %10lf, J = %10lf\n",ngrad2,J);

  for (i=0;;i++) {
    getNgrad2(&ngrad2);
    if (ngrad2<1e-3) break;
    copyXx2X();
  }
  printf("After %d iterations:\n",i);
  getX(x);
  for (i=0;i<n;i++)
    printf("   x[%3d] = %10lf\n",i,x[i]);
  getNgrad2(&ngrad2);
  getJ(J);
  printf("   ngrad2 = %10lf, J = %10lf\n",ngrad2,J);
}
```

The following matlab code uses cmexTools to create the cmex function and then calls it:

```
N=100;
n=8;
```

```
b=rand(N,1);

createGateway('template','docCmex.c',... % template describing the cmex function
              'compile',true,...
              'preprocessParameters',{N,n,b});

A=rand(N,n);
x0=rand(n,1);

[J,x]=tmpC_docCmex(A,x0);
```

4.3 Matlab class

Alternatively[4], by using

```
compile2matlab(code,'tmpM_testDoc.m');
```

instead of the above `compile2C` command, one can generate a MATLAB® class to perform the same computions from within MATLAB®, e.g., using the following code:

```
obj=tmpC_testDoc();
setA(obj,rand(N,n));
x=zeros(n,1);
setX(obj,x);
 while 1
   ngrad=getNgrad(obj);
   if sqrt(ngrad)<1e-3, break; end
   copyXx2X(obj);
 end
j=getJ(obj);
```

[4] Not (yet?) implemented.