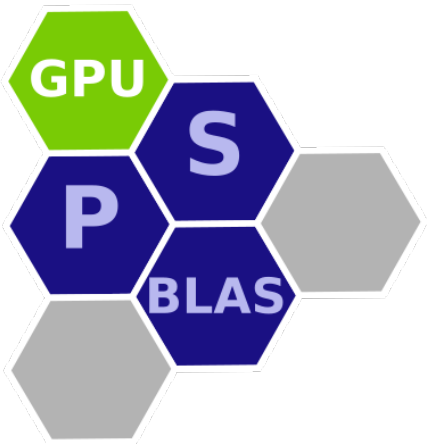
PSBLAS-Extensions 1.0

A reference guide for the Parallel Sparse BLAS library



by Salvatore Filippone
University of Rome "Tor Vergata".

March 25, 2015.

Contents

1 Introduction

The PSBLAS-EXT library contains a set of extensions to the base library. The extensions provide additional storage formats beyond the ones already contained in the base library, as well as interfaces to two external libraries:

SPGPU https://code.google.com/p/spgpu/, for computations on NVIDIA GPUs;

LIBRSB http://sourceforge.net/projects/librsb/, for computations on multicore parallel machines.

The infrastructure laid out in the base library to allow for these extensions is detailed in the references [2, 4]; the GPU-specific data formats are described in [1].

1.1 Application structure

A sample application using the PSBLAS extensions will contain the following steps:

- `USE` the appropriat modules (`psb_ext_mod`, `psb_gpu_mod`);
- Declare a *mold* variable of the necessary type (e.g. `psb_d_ell_sparse_mat`, `psb_d_hlg_sparse_mat`, `psb_d_vect_gpu`);
- Pass the mold variable to the base library interface where needed to ensure the appropriate dynamic type.

Suppose you want to use the GPU-enabled ELLPACK data structure; you would use a piece of code like this (and don't forget, you need GPU-side vectors along with the matrices):

```
program my_gpu_test
  use psb_base_mod
  use psb_util_mod
  use psb_ext_mod
  use psb_gpu_mod
  type(psb_dspmat_type) :: a, agpu
  type(psb_d_vect_type) :: x, xg, bg

  real(psb_dpk_), allocatable :: xtmp(:)
  type(psb_d_vect_gpu)       :: vmold
  type(psb_d_elg_sparse_mat) :: aelg
  type(psb_ctxt_type) :: ctxt
  integer             :: iam, np


  call psb_init(ctxt)
  call psb_info(ctxt,iam,np)
  call psb_gpu_init(ctxt, iam)


  ! My own home-grown matrix generator
```

```
  call gen_matrix(ctxt,idim,desc_a,a,x,info)
  if (info /= 0) goto 9999

  call a%cscnv(agpu,info,mold=aelg)
  if (info /= 0) goto 9999
  xtmp = x%get_vect()
  call xg%bld(xtmp,mold=vmold)
  call bg%bld(size(xtmp),mold=vmold)

  ! Do sparse MV
  call psb_spmm(done,agpu,xg,dzero,bg,desc_a,info)


9999 cotinue
  if (info == 0) then
     write(*,*) '42'
  else
     write(*,*) 'Something went wrong ',info
  end if


  call psb_gpu_exit()
  call psb_exit(ctxt)
  stop
end program my_gpu_test
```

A full example of this strategy can be seen in the `test/ext/kernel` and `test/gpu/kernel` subdirectories, where we provide sample programs to test the speed of the sparse matrix-vector product with the various data structures included in the library.

2 Data Structures

Access to the facilities provided by `psblas-ext` is mainly through the data types that are provided within. The data classes are derived from the base classes in PSBLAS, through the Fortran 2003 mechanism of *type extension* [6]. The data classes are divided between the general purpose CPU extensions, the GPU interfaces and the RSB interfaces.
In the description we will make use of the notation introduced in Table 1.

Table 1: Notation for parameters describing a sparse matrix

Name	Description
M	Number of rows in matrix
N	Number of columns in matrix
NZ	Number of nonzeros in matrix
AVGNZR	Average number of nonzeros per row
MAXNZR	Maximum number of nonzeros per row
NDIAG	Numero of nonzero diagonals
AS	Coefficients array
IA	Row indices array
JA	Column indices array
IRP	Row start pointers array
JCP	Column start pointers array
NZR	Number of nonzeros per row array
OFFSET	Offset for diagonals

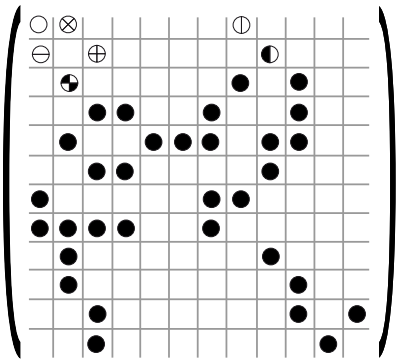


Figure 1: Example of sparse matrix

2.1 CPU-class extensions

ELLPACK

The ELLPACK/ITPACK format (shown in Figure 2) comprises two 2-dimensional arrays `AS` and `JA` with `M` rows and `MAXNZR` columns, where `MAXNZR` is the maximum number of nonzeros in any row [**?**]. Each row of the arrays `AS` and `JA` contains the coefficients and column indices; rows shorter than `MAXNZR` are padded with zero coefficients and appropriate column indices, e.g. the last valid one found in the same row.

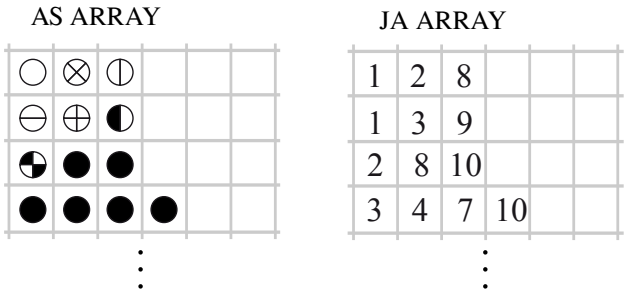


Figure 2: ELLPACK compression of matrix in Figure 1

```
do i=1,n
  t=0
  do j=1,maxnzr
    t = t + as(i,j)*x(ja(i,j))
  end do
  y(i) = t
end do
```

Algorithm 1: Matrix-Vector product in ELL format

The matrix-vector product $y = Ax$ can be computed with the code shown in Alg. 1; it costs one memory write per outer iteration, plus three memory reads and two floating-point operations per inner iteration.
Unless all rows have exactly the same number of nonzeros, some of the coefficients in the `AS` array will be zeros; therefore this data structure will have an overhead both in terms of memory space and redundant operations (multiplications by zero). The overhead can be acceptable if:

1. The maximum number of nonzeros per row is not much larger than the average;
2. The regularity of the data structure allows for faster code, e.g. by allowing vectorization, thereby offsetting the additional storage requirements.

In the extreme case where the input matrix has one full row, the ELLPACK structure would require more memory than the normal 2D array storage. The ELLPACK storage format was very popular in the vector computing days; in modern CPUs it is not quite as popular, but it is the basis for many GPU formats.
The relevant data type is `psb_T_ell_sparse_mat`:

```
type, extends(psb_d_base_sparse_mat) :: psb_d_ell_sparse_mat
  !
  ! ITPACK/ELL format, extended.
  !

  integer(psb_ipk_), allocatable :: irn(:), ja(:,:), idiag(:)
  real(psb_dpk_), allocatable :: val(:,:)
```

```
contains
  ....
end type psb_d_ell_sparse_mat
```

Hacked ELLPACK

The *hacked ELLPACK* (**HLL**) format alleviates the main problem of the ELLPACK format, that is, the amount of memory required by padding for sparse matrices in which the maximum row length is larger than the average. The number of elements allocated to padding is $[(m * maxNR) - (m * avgNR) = m * (maxNR - avgNR)]$ for both `AS` and `JA` arrays, where m is equal to the number of rows of the matrix, $maxNR$ is the maximum number of nonzero elements in every row and $avgNR$ is the average number of nonzeros. Therefore a single densely populated row can seriously affect the total size of the allocation.
To limit this effect, in the HLL format we break the original matrix into equally sized groups of rows (called *hacks*), and then store these groups as independent matrices in ELLPACK format. The groups can be arranged selecting rows in an arbitrarily manner; indeed, if the rows are sorted by decreasing number of nonzeros we obtain essentially the JAgged Diagonals format. If the rows are not in the original order, then an additional vector *rIdx* is required, storing the actual row index for each row in the data structure.
The multiple ELLPACK-like buffers are stacked together inside a single, one dimensional array; an additional vector *hackOffsets* is provided to keep track of the individual submatrices. All hacks have the same number of rows *hackSize*; hence, the *hackOffsets* vector is an array of $(m/hackSize) + 1$ elements, each one pointing to the first index of a submatrix inside the stacked cM/rP buffers, plus an additional element pointing past the end of the last block, where the next one would begin. We thus have the property that the elements of the k-th *hack* are stored between `hackOffsets[k]` and `hackOffsets[k+1]`, similarly to what happens in the CSR format.

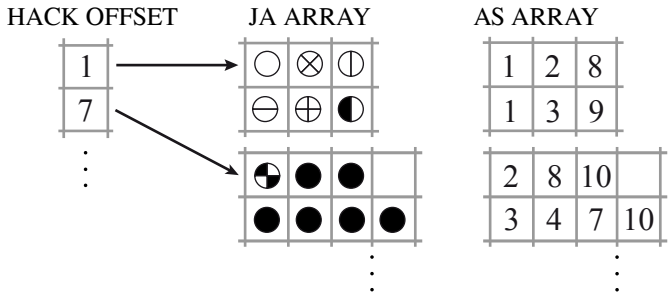


Figure 3: Hacked ELLPACK compression of matrix in Figure 1

With this data structure a very long row only affects one hack, and therefore the additional memory is limited to the hack in which the row appears.
The relevant data type is `psb_T_hll_sparse_mat`:

```
type, extends(psb_d_base_sparse_mat) :: psb_d_hll_sparse_mat
  !
  ! HLL format. (Hacked ELL)
  !
  integer(psb_ipk_) :: hksz
  integer(psb_ipk_), allocatable :: irn(:), ja(:), idiag(:),
   ↪  hkoffs(:)
  real(psb_dpk_), allocatable :: val(:)

contains
 ....
end type
```

Diagonal storage

The DIAgonal (DIA) format (shown in Figure 4) has a 2-dimensional array AS containing in each column the coefficients along a diagonal of the matrix, and an integer array OFFSET that determines where each diagonal starts. The diagonals in AS are padded with zeros as necessary.
The code to compute the matrix-vector product $y = Ax$ is shown in Alg. 2; it costs one memory read per outer iteration, plus three memory reads, one memory write and two floating-point operations per inner iteration. The accesses to AS and x are in strict sequential order, therefore no indirect addressing is required.

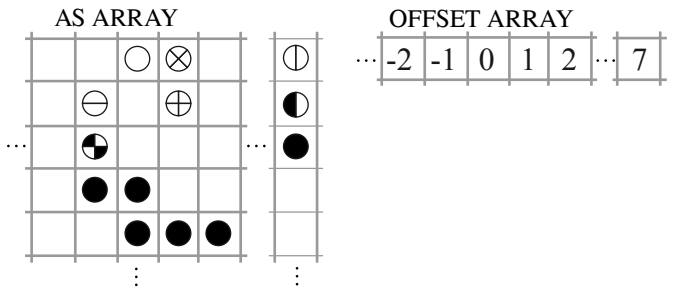


Figure 4: DIA compression of matrix in Figure 1

The relevant data type is psb_T_dia_sparse_mat:

```
type, extends(psb_d_base_sparse_mat) :: psb_d_dia_sparse_mat
  !
  ! DIA format, extended.
  !

  integer(psb_ipk_), allocatable :: offset(:)
  integer(psb_ipk_) :: nzeros
  real(psb_dpk_), allocatable :: data(:,:)

end type
```

```
do j=1,ndiag
  if (offset(j) > 0) then
    ir1 = 1; ir2 = m - offset(j);
  else
    ir1 = 1 - offset(j); ir2 = m;
  end if
  do i=ir1,ir2
    y(i) = y(i) + alpha*as(i,j)*x(i+offset(j))
  end do
end do
```

Algorithm 2: Matrix-Vector product in DIA format

Hacked DIA

Storage by DIAgonals is an attractive option for matrices whose coefficients are located on a small set of diagonals, since they do away with storing explicitly the indices and therefore reduce significantly memory traffic. However, having a few coefficients outside of the main set of diagonals may significantly increase the amount of needed padding; moreover, while the DIA code is easily vectorized, it does not necessarily make optimal use of the memory hierarchy. While processing each diagonal we are updating entries in the output vector y, which is then accessed multiple times; if the vector y is too large to remain in the cache memory, the associated cache miss penalty is paid multiple times.
The *hacked DIA* (**HDIA**) format was designed to contain the amount of padding, by breaking the original matrix into equally sized groups of rows (*hacks*), and then storing these groups as independent matrices in DIA format. This approach is similar to that of HLL, and requires using an offset vector for each submatrix. Again, similarly to HLL, the various submatrices are stacked inside a linear array to improve memory management. The fact that the matrix is accessed in slices helps in reducing cache misses, especially regarding accesses to the vector y.
An additional vector *hackOffsets* is provided to complete the matrix format; given that *hackSize* is the number of rows of each hack, the *hackOffsets* vector is made by an array of $(m/hackSize) + 1$ elements, pointing to the first diagonal offset of a submatrix inside the stacked *offsets* buffers, plus an additional element equal to the number of nonzero diagonals in the whole matrix. We thus have the property that the number of diagonals of the *k*-th *hack* is given by *hackOffsets[k+1] - hackOffsets[k]*.
The relevant data type is `psb_T_hdia_sparse_mat`:

```
type pm
   real(psb_dpk_), allocatable  :: data(:,:)
end type pm

type po
   integer(psb_ipk_), allocatable  :: off(:)
end type po
```

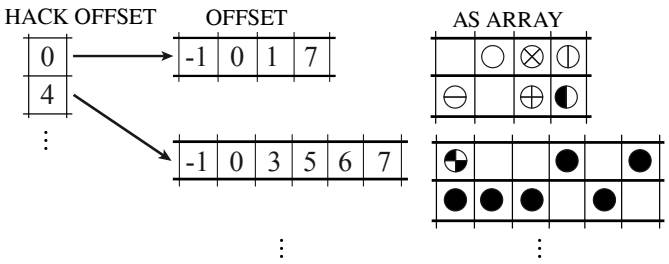


Figure 5: Hacked DIA compression of matrix in Figure 1

```
type, extends(psb_d_base_sparse_mat) :: psb_d_hdia_sparse_mat
  !
  ! HDIA format, extended.
  !

  type(pm), allocatable :: hdia(:)
  type(po), allocatable :: offset(:)
  integer(psb_ipk_) :: nblocks, nzeros
  integer(psb_ipk_) :: hack = 64
  integer(psb_long_int_k_) :: dim=0

contains
 ....
end type
```

2.2 GPU-class extensions

For computing on the GPU we define a dual memorization strategy in which each variable on the CPU ("host") side has a GPU ("device") side. When a GPU-type variable is initialized, the data contained is (usually) the same on both sides. Each operator invoked on the variable may change the data so that only the host side or the device side are up-to-date.
Keeping track of the updates to data in the variables is essential: we want to perform most computations on the GPU, but we cannot afford the time needed to move data between the host memory and the device memory because the bandwidth of the interconnection bus would become the main bottleneck of the computation. Thus, each and every computational routine in the library is built according to the following principles:

- If the data type being handled is GPU-enabled, make sure that its device copy is up to date, perform any arithmetic operation on the GPU, and if the data has been altered as a result, mark the main-memory copy as outdated.
- The main-memory copy is never updated unless this is requested by the user either

  explicitly by invoking a synchronization method;

implicitly by invoking a method that involves other data items that are not GPU-enabled, e.g., by assignment ov a vector to a normal array.

In this way, data items are put on the GPU memory "on demand" and remain there as long as "normal" computations are carried out. As an example, the following call to a matrix-vector product

```
call psb_spmm(alpha,a,x,beta,y,desc_a,info)
```

will transparently and automatically be performed on the GPU whenever all three data inputs `a`, `x` and `y` are GPU-enabled. If a program makes many such calls sequentially, then

- The first kernel invocation will find the data in main memory, and will copy it to the GPU memory, thus incurring a significant overhead; the result is however *not* copied back, and therefore:
- Subsequent kernel invocations involving the same vector will find the data on the GPU side so that they will run at full speed.

For all invocations after the first the only data that will have to be transferred to/from the main memory will be the scalars `alpha` and `beta`, and the return code `info`.

Vectors: The data type `psb_T_vect_gpu` provides a GPU-enabled extension of the inner type `psb_T_base_vect_type`, and must be used together with the other inner matrix type to make full use of the GPU computational capabilities;

CSR: The data type `psb_T_csrg_sparse_mat` provides an interface to the GPU version of CSR available in the NVIDIA CuSPARSE library;

HYB: The data type `psb_T_hybg_sparse_mat` provides an interface to the HYB GPU storage available in the NVIDIA CuSPARSE library. The internal structure is opaque, hence the host side is just CSR;

ELL: The data type `psb_T_elg_sparse_mat` provides an interface to the ELLPACK implementation from SPGPU;

HLL: The data type `psb_T_hlg_sparse_mat` provides an interface to the Hacked ELLPACK implementation from SPGPU;

HDIA: The data type `psb_T_hdiag_sparse_mat` provides an interface to the Hacked DIAgonals implementation from SPGPU;

3 GPU Environment Routines

psb_gpu_init — Initializes PSBLAS-GPU environment

```
call psb_gpu_init(ctxt [, device])
```

This subroutine initializes the PSBLAS-GPU environment.

Type: Synchronous.

On Entry

device ID of GPU device to attach to.
Scope: **local**.
Type: **optional**.
Intent: **in**.
Specified as: an integer value. Default: use `mod(iam,ngpu)` where `iam` is the calling process index and `ngpu` is the total number of GPU devices available on the current node.

Notes

1. A call to this routine must precede any other PSBLAS-GPU call.

psb_gpu_exit — Exit from PSBLAS-GPU environment

```
call psb_gpu_exit(ctxt)
```

This subroutine exits from the PSBLAS GPU context.

Type: Synchronous.

On Entry

ctxt the communication context identifying the virtual parallel machine.
Scope: **global**.
Type: **required**.
Intent: **in**.
Specified as: an integer variable.

psb_gpu_DeviceSync — Synchronize GPU device

```
call psb_gpu_DeviceSync()
```

This subroutine ensures that all previosly invoked kernels, i.e. all invocation of GPU-side code, have completed.

psb_gpu_getDeviceCount

```
ngpus =  psb_gpu_getDeviceCount()
```

Get number of devices available on current computing node.

psb_gpu_getDevice

```
ngpus =  psb_gpu_getDevice()
```

Get device in use by current process.

psb_gpu_setDevice

```
info = psb_gpu_setDevice(dev)
```

Set device to be used by current process.

psb_gpu_DeviceHasUVA

```
hasUva = psb_gpu_DeviceHasUVA()
```

Returns true if device currently in use supports UVA (Unified Virtual Addressing).

psb_gpu_WarpSize

```
nw = psb_gpu_WarpSize()
```

Returns the warp size.

psb_gpu_MultiProcessors

```
nmp = psb_gpu_MultiProcessors()
```

Returns the number of multiprocessors in the GPU device.

psb_gpu_MaxThreadsPerMP

```
nt = psb_gpu_MaxThreadsPerMP()
```

Returns the maximum number of threads per multiprocessor.

psb_gpu_MaxRegistersPerBlock

```
nr = psb_gpu_MaxRegistersPerBlock()
```

Returns the maximum number of register per thread block.

psb_gpu_MemoryClockRate

```
cl = psb_gpu_MemoryClockRate()
```

Returns the memory clock rate in KHz, as an integer.

psb_gpu_MemoryBusWidth

```
nb = psb_gpu_MemoryBusWidth()
```

Returns the memory bus width in bits.

psb_gpu_MemoryPeakBandwidth

```
bw = psb_gpu_MemoryPeakBandwidth()
```

Returns the memory peak bandwidth in MB/s (real double precision).

References

[1] D. Barbieri, V. Cardellini, A. Fanfarillo, S. Filippone, Three storage formats for sparse matrices on GPGPUs, Tech. Rep. DICII RR-15.6, Università di Roma Tor Vergata (February 2015).

[2] Cardellini, V., Filippone, S., and Rouson, D. 2014, Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms, *Scientific Programming 22*, 1, 1–19.

[3] S. Filippone and M. Colajanni, *PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices*, ACM Transactions on Mathematical Software, 26(4), pp. 527–550, 2000.

[4] S. Filippone and A. Buttari, *Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003*, ACM Transactions on Mathematical Software, 38(4), 2012.

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[6] Metcalf, M., Reid, J. and Cohen, M. *Modern Fortran explained.* Oxford University Press, 2011.