Cache and Energy Efficiency of Sparse Matrix-Vector Multiplication for different BLAS Numerical Types with the RSB Format

Michele Martone

High Level Support Team
Max Planck Institute for Plasma Physics
Garching bei Muenchen, Germany

PARCO'13
Munich, Germany
September 13, 2013

Presentation outline

Goal of this Study

- quantify and relate energy, cache usage and time savings of `librsb`'s RSB over Intel's MKL[1] CSR for SParse Matrix-Vector multiply (*SpMV*) for matrices of an example application
- ... for different numerical types

---

[1] Math Kernel Library

Context: Sparse Matrix Computations

- numerical matrices which are *large* and populated mostly by zeros
- ubiquitous in scientific/engineering computations (e.g.: PDE)
- the *performance* of sparse matrix codes computation on modern CPUs can be problematic (a fraction of peak)!

Context: The four *Basic Linear Algebra Subroutines* (BLAS) numerical types

For each, its occupation (`sizeof()`) S in bytes:

- D: *double precision real*
  $S_D = 8$
- Z: *double precision complex*
  $S_Z = 16$
- S: *single precision real*
  $S_S = 4$
- C: *single precision complex*
  $S_C = 8$

Matrix representations that matter to us

- *coordinate* (COO): used mostly in matrix specification
- *compressed sparse rows* (CSR): used often in computations

In most common implementations (e.g.: Intel's MKL), 4 byte integers are used for COO/CSR indices types.

Basic representation: Coordinate (COO)

$$A = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 \\ 0 & 0 & a_{3,3} & 0 \\ 0 & 0 & 0 & a_{4,4} \end{vmatrix}$$

- $VA = [a_{1,1}, a_{1,2}, a_{1,3}, a_{2,2}, a_{2,3}, a_{3,3}, a_{4,4}]$ (*nonzeroes*)
- $IA = [1, 1, 1, 2, 2, 3, 4]$ (*nonzeroes row indices*)
- $JA = [1, 2, 3, 2, 3, 3, 4]$ (*nonzeroes column indices*)
- so, $a_{i,j} = VA(n) \;\; iff \;\; IA(n) = i, JA(n) = j$
- occupation for type T: $nnz \cdot (S_T + 4 + 4)$

Standard representation: Compressed Sparse Rows (CSR)

$$A = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 \\ 0 & 0 & a_{3,3} & 0 \\ 0 & 0 & 0 & a_{4,4} \end{vmatrix}$$

- $VA = [a_{1,1}, a_{1,2}, a_{1,3}, a_{2,2}, a_{2,3}, a_{3,3}, a_{4,4}]$ (*nonzeroes*)
- $JA = [1, 2, 3, 2, 3, 3, 4]$ (*nonzeroes column indices*)
- $RP = [1, 4, 6, 7, 8]$ (*row pointers, for each row*)
- so, elements on line i are in positions $VA(RP(i))$ *to* $VA(RP(i+1)) - 1$
- so, $a_{i,j} = VA(n)$ *iff* $JA(n) = j$
- occupation for type T: $nnz \cdot (S_T + 4) + 4 \cdot nrows$

Pros and Cons of CSR in a nutshell

- + common, easy to work with
- + parallel *SpMV* is feasible
- - parallel *SpMV-T* is feasible ...but poor performance
- - the above are relatively inefficient with *large*[2] matrices
- - impractical for parallel sparse *triangular solve*

---

[2] Matrices that don't fit in the cache memory.

A recursive matrix storage: *Recursive Sparse Blocks* (RSB)

we propose:

- a *quad-tree* of sparse *leaf* submatrices
- outcome of recursive *partitioning* in *quadrants*
- leaf submatrices are stored by either *row oriented Compressed Sparse Rows* (CSR) or *Coordinates* (COO)
- an *unified* format for Sparse **BLAS**[3] operations and variations (e.g.: diagonal implicit, one or zero based indices, transposition, complex types, stride, ...)
- partitioning with regards to both the underlying cache size **and** available threads
- leaf submatrices are *cache blocks*

---

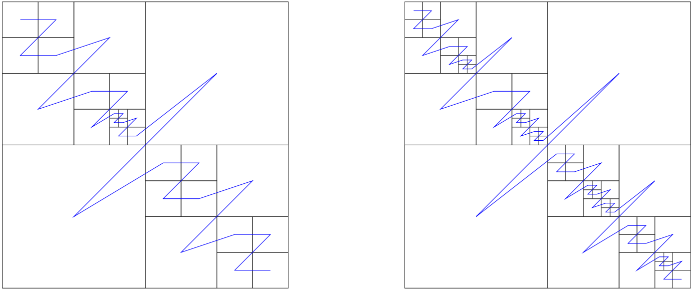
[3] Sparse Basic Linear Algebra Subprograms standard, as in TOMS Algorithm 818 (Duff and Vömel, 2002).

Design goals of `librsb` and the RSB format

- parallel, efficient *SpMV*/triangular solve/*COO* → *RSB*
- in-place *COO* ↔ *RSB* conversions
- no oversized COO arrays / no *fill-in* (e.g.: in contrast to BCSR)
- no need to pad x, y vectors arrays with extra elements
- developed on/for shared memory cache based CPUs:
  - *locality of memory references*
  - coarse-grained *workload partitioning*
- architecture independent (C'99, POSIX, OpenMP)
- `librsb` is available as *free software* on SourceForge

Adaptivity to Cache Size

Sample matrix from our application (a *small* one). Each block should occupy approximately the same amount of memory.



On the left, blocking for S type; on the right, for Z.

Adaptivity to threads count

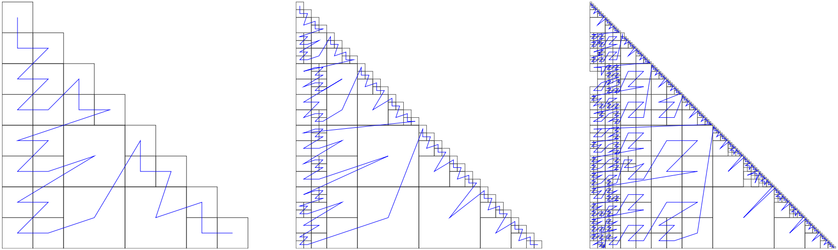


Figure: Matrix *audikw_1* (symmetric, 943695 rows, $3.9 \cdot 10^7$ nonzeroes) for 1, 4 and 16 threads on a Sandy Bridge.

Memory Occupation of CSR and RSB

- CSR's is fixed:

  $$nnz \cdot (S+4) + nrows \cdot \; 4$$

- `librsb` RSB's varies between:

  $$nnz \cdot (S+2) + nrows \cdot \; 4$$

  and

  $$nnz \cdot (S+8)$$

Occupation of RSB w.r.t CSR

For the different types:

Δ / type	S	D/C	Z
min	-25%	-16%	-10%
max	+50%	+33%	+20%

(approximately)

Impact of Matrix Memory Occupation

- it can influence run-time (*SpMV*) accessed memory
- run-time accessed memory is what matters
- it's better if the access pattern leads to less cache traffic

Experimental Setup (1)

- Matrices resulting from the description of global, resistive, linear MHD (Magnetohydrodynamics) studied in toroidal geometry (see Bondeson and Vlad, 1992).
  We concentrate on the largest: $9.62 \cdot 10^7$ nonzeroes, $1.99 \cdot 10^5$ equations (with an average of 484 nonzeroes per row).[4]
- On a 2 x "Sandy Bridge E5-2670"; L3: 20MB, L2: 256KB, L1:32KB
- We instrument the code with the LIKWID *performance tool* (Treibig, Hager, Wellein'2011) to collect "`ENERGY`" and "`L2 data volume`" metrics
- We report:
  - performance in canonical GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$)
  - spent energy in kJ/GFlop
  - L2 traffic in bytes/nonzero

---

[4] Results are similar for smaller matrices, as long as outermost cache size is exceeded.

Experimental Setup (2)

- Intel C Compiler
- `CFLAGS=-O3 -fPIC -restrict -openmp`
- `mkl_dcsrmv`, `mkl_zcsrmv`, `mkl_ccsrmv`, `mkl_scsrmv` from *"MKL 11.0-1, Product, 20121009 ..."*
- no memory placement tool, no clock control

Serial Results: S

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
CSR/S/1	2.31	-7.95	28.08	+6.71	8.55	+2.65
MKL/S/1	2.51	0.00	26.31	0.00	8.33	0.00
RSB/S/1	2.33	-7.03	27.11	+3.01	6.19	-25.71

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Serial Results: *D*

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
CSR/D/1	1.81	-5.25	36.40	+2.94	12.20	-0.10
MKL/D/1	1.91	0.00	35.36	0.00	12.21	0.00
RSB/D/1	1.84	-3.73	35.10	-0.74	10.20	-16.50

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Serial Results: *C*

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
CSR/C/1	3.55	-50.54	17.66	+87.77	13.87	+0.70
MKL/C/1	7.19	0.00	9.40	0.00	13.78	0.00
RSB/C/1	3.53	-50.89	17.52	+86.32	11.44	-16.98

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Serial Results: Z

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
CSR/Z/1	2.68	-30.29	23.83	+37.23	49.43	-4.19
MKL/Z/1	3.85	0.00	17.37	0.00	51.59	0.00
RSB/Z/1	2.69	-30.12	23.69	+36.39	34.41	-33.29

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Parallel Results: *S*

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
MKL/S/12	10.13	0.00	25.47	0.00	8.26	0.00
MKL/S/16	8.26	0.00	35.39	0.00	8.25	0.00
RSB/S/12	14.48	+43.01	17.98	-29.43	6.18	-25.15
RSB/S/16	13.15	+59.07	24.34	-31.21	6.30	-23.66

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Parallel Results: *D*

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
MKL/D/12	7.18	0.00	35.15	0.00	12.22	0.00
MKL/D/16	5.50	0.00	51.99	0.00	12.22	0.00
RSB/D/12	8.61	+19.95	29.58	-15.83	10.33	-15.43
RSB/D/16	8.50	+54.61	35.79	-31.16	10.35	-15.31

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Parallel Results: *C*

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
MKL/C/12	28.79	0.00	9.03	0.00	13.80	0.00
MKL/C/16	21.62	0.00	13.52	0.00	13.81	0.00
RSB/C/12	32.27	+12.08	8.40	-6.92	10.43	-24.43
RSB/C/16	31.77	+46.93	9.94	-26.53	10.44	-24.44

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Parallel Results: *Z*

Implem.	Speed	%Δ to MKL	Energy	%Δ to MKL	L2 Traffic	%Δ to MKL
MKL/Z/12	17.21	0.00	15.20	0.00	49.61	0.00
MKL/Z/16	13.09	0.00	22.45	0.00	49.62	0.00
RSB/Z/12	19.36	+12.50	13.59	-10.64	19.30	-61.08
RSB/Z/16	18.25	+39.44	17.07	-23.96	19.38	-60.94

Table: Metrics: execution Speed in (canonical) GFlops ($2 \cdot 10^{-9} \cdot nnz \cdot elapsed\_seconds^{-1}$; 3 times that for complex); Energy in kJ/GFlop; L2 Traffic in bytes/nnz.

Conclusions, serial runs

- serially, RSB is slower than MKL by respectively:

  3.73% (D), 30.12% (Z), 7.03% (S), 50.89% (C)
- with no bandwith limitations, MKL's optimized serial kernels are better!

Conclusions, parallel runs

- RSB results were better than MKL's by respectively:

  19.95% (D), 12.50% (Z), 43.01% (S), 12.08% (C)
- the *energy-cheapest Flops* were associated to the fastest executions, confirming e.g.: (Hager et al., 2012)
- 12-threaded performed better than 16-threaded!
- energy savings over MKL were roughly half the savings in speed

Further Work

extend study to...

- *auto-tuning*: locating *best core count* and *best subdivision*
- other operations (symmetric multiply, transposed multiply, conversion, ...)
- other matrices
- compilers impact on *bandwidth limited* RSB kernels

References

- Sparse BLAS: Iain S. Duff and Christof Vömel. *Algorithm 818: A reference model implementation of the Sparse BLAS in Fortran 95* In ACM Trans. on Math. Softw., n. 2, vol. 28, pages 268–283, ACM, 2002

- `librsb`: `http://sourceforge.net/projects/librsb`
- RSB: Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. *Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations.* In Thomas Philips, editor, CATA, pages 300-305. ISCA, 2010.
- Our sample application: A. Bondeson, G. Vlad, et al.. *Resistive toroidal stability of internal kink modes in circular and shaped tokamaks.* In Physics of Fluids B: Plasma Physics, 4(7):1889–1900, 1992.

Acknowledgements

I wish to thank my colleagues at HLST for providing me with criticism with this presentation.

Questions / discussion welcome!

Thanks for your attention.

Please consider using `librsb`:
`http://sourceforge.net/projects/librsb`

Extra: Minimal Memory Occupation of RSB

at most, RSB saves:
$\frac{4\ nnz+4\ nrows-2\ nnz-4\ nrows}{4\ nnz+4\ nrows+nnzS} = \frac{2\ nnz}{4\ nnz+4\ nrows+nnzS} \approx \frac{2\ nnz}{4\ nnz+nnzS} = \frac{2}{4+S}$
over CSR:
this is $1/4$ for$S = S_S$, $1/6$ for S_D/S_C, $1/10$ for S_Z.

Extra: Maximal Memory Occupation of RSB

at most, RSB uses:

$$\frac{8\ nnz - 4\ nnz - 4\ nrows}{4\ nnz + 4\ nrows + nnz S} = \frac{4\ nnz - 4\ nrows}{4\ nnz + 4\ nrows + nnz S} \approx \frac{4\ nnz}{4\ nnz + nnz S} = \frac{4}{4+S}$$

more than CSR:

this is $1/2$ for S_S, $1/3$ for S_D/S_C, $1/5$ for S_Z.