Optimizing Sparse Matrix-Vector Multiplication via Dynamic Reigister Blocking and Reordered Cache Blocking

Nithin Tumma, Neel Patel

CS265, Harvard University, Fall 2014

HARVARD School of Engineering and Applied Sciences

Abstract

The Sparse Matrix-Vector multiplication (SpMV) kernel is of central importance in scientific computing, with applications ranging from PDE solvers to social network analysis. Due to random accesses into the source vector, naive implementations of SpMV based on the compressed sparse row (CSR) format have very high memory cost (cache misses, TLB misses, etc.), and in many cases the CPU is only utilized at 10\% of its peak operating performance. Prior work has focused on exploiting dense substructures within a sparse matrix via blocking the matrix into non-overlapping r x c subblocks. However, because the block size is static, this method wastes a lot of space/computation. We propose a new storage format and SpMV algorithm that supports dynamic block sizes. Then, we develop a pre-processing technique to choose the proper block size (based on local density) for each non-zero in the matrix. To further improve cache locality, we also develop a pre-processing step to optimize the order in which the blocks-vector products are computed (i.e., reordering a cache block). We compare our approach to an implementation of Sparsity, a leading SpMV optimization framework using static register block sizes and cache blocking. Our dynamic blocking approach is shown to significantly reduce storage space (2.7x on average across test matrices) compared to static blocking. Furthermore, compared to Sparsity, our SpMV compute time is decreased by roughly the same magnitude. Memory profiling shows our methods significantly reduce cache and TLB misses. Moreover, our pre-processing time is within 5x the pre-processing time of Sparsity's, easily amortizable over repeated multiplications.

Introduction and Background

Sparse matrices are amenable to optimizations to the representation and storage of data, but traditional implementations of sparse matrix-vector multiplication have been relatively poor [Williams].

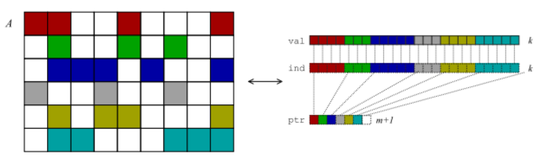
We also see many cases of problems where an offline matrix needs to be multiplied by many vectors throughout the day (ie in predicting a recommendation for a variable user [online] based on a static sparse matrix of data). Because we have one offline and unchanging matrix, we can precompute and preprocess this matrix once, and have the cost be amortized over many computations throughout the day.

Our work optimizes sparse matrix-vector multiplication through:

- **Dynamic blocking** – dividing the sparse matrix into dense (and variably sized) rectangular blocks small enough to fit in register
- **Reorded Cache Blocking**– groups of blocks stored contiguously on pages in memory with read/write locality

Storage

A common storage form for sparse matrices is **compressed sparse row** (CSR) format.



Storage

There is also an adaptation of CSR called **blocked compressed row storage** (BCRS). This format exploits dense blocks in a matrix to store data in a more compressed format.

We modified BSRS to allow dynamic blocking (i.e., storing a block identifier for each block to determine its shape).

$$B_{r\times c} = \underbrace{\boxed{\mathrm{B}_i}}_{\text{block id}}, \overbrace{\boxed{\mathrm{R}_i}\boxed{\mathrm{C}_i}}^{\text{row, col index}}, \underbrace{\boxed{\mathrm{A}[\mathrm{R}_i, C_i]}\boxed{\mathrm{A}[\mathrm{R}_i+1, C_i]}\ldots\boxed{\mathrm{A}[\mathrm{R}_i+(r-1), C_i+(c-1)]}}_{\text{values}}$$

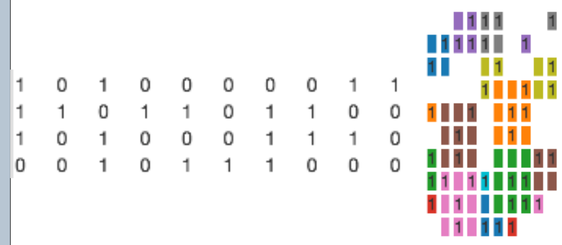
Test Matrix Suite

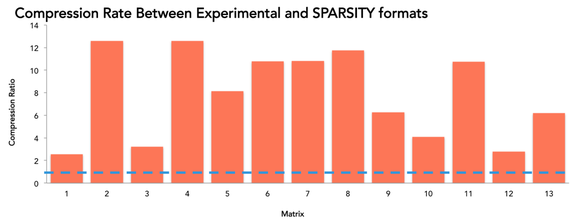
ID	Matrix Name	Matrix Image	Description	# Rows (adj)	# Cols (adj)	# Non-Zeros	Sparsity (%)
1	Tina_AskCal		Pajek network of a student government	12	12	29	20.139%
2	lp_pds_02		Netlib LP problem pds_2, minimization with constraints	2,964	7,716	16,571	0.072%
3	rail516		H. Mittelmann test set on Italian railways	516	47,832	315,412	1.278%
4	lp_pds_10		Netlib LP problem pds_10, minimization with constraints	16,560	49,932	107,605	0.013%
5	TF17		Forests and Trees from Nicolas Thiery	38,136	48,636	586,218	0.032%
6	m2010		A graph of census blocks in New Hampshire, 2010	48,840	48,840	234,550	0.010%
7	wy2010		A graph of census blocks in Wyoming, 2010	86,208	86,208	427,586	0.006%
8	ut2010		A graph of census blocks in Utah, 2010	115,416	115,416	286,033	0.002%
9	shar_te2-b3		Simplicial complexes from Homology from Volkmar Welker	200,208	200,208	200,200	0.00050%
10	ch7-9-b5		Simplicial complexes from Homology from Volkmar Welker	423,360	317,520	2,540,160	0.002%
11	l2010		A graph of census blocks in Illinois, 2010	451,560	451,560	1,082,232	0.001%
12	m2depi		2D Markov model of epidemic	525,828	525,828	2,100,225	0.001%
13	roadNet-PA		Road network of Pennsylvania	1,090,920	1,090,920	3,083,796	0.00026%

Dynamic Register Blocking

Based on the size of a system's register, we can determine *r_max* and *c_max*, the maximum dimensions of a block that can fit in register (along with space for the source vector and for writing output). Then, our dynamic register blocking scheme is based on sampling the region around each non-zero, and choosing the largest block size with sufficient density that covers the original non-zero.

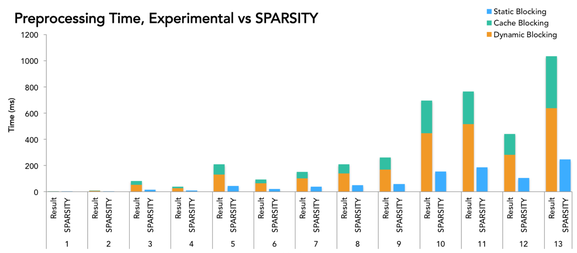
- We allowed 1x1, 2x2, 3x3, and 4x4 register blocks and achieved significant compression compared to static blocking.





Reordering Cache Blocks

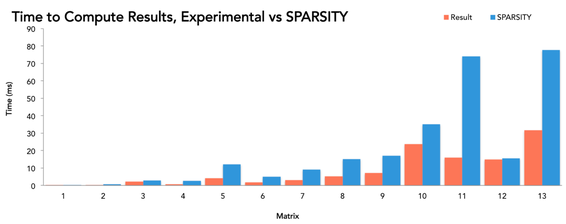
We cache block in the same manner as Sparsity, but within a ache block, we order the blocks to force successive reads/writes to the same cache lines (if possible).



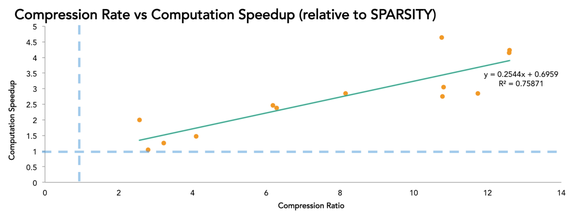
- Our pre-processing time is always within **5x** of Sparsity.
- Each Cache-block is stored as a contiguous sequence of register blocks, as below.

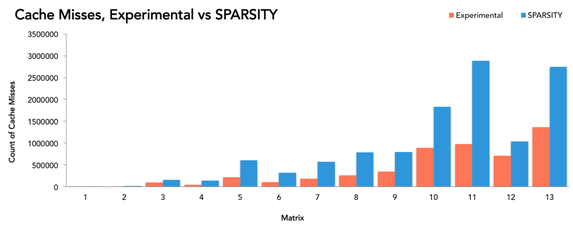
$$S_i = \boxed{\mathrm{A}_1}\boxed{\mathrm{A}_2}\cdots\boxed{\mathrm{A}_k}$$

SpMV Compute Performance

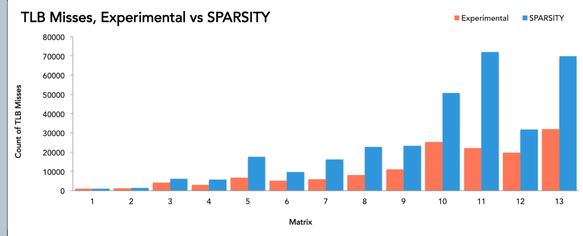


- Average speedup of **2.71x** compared to Sparsity





SpMV Compute Performance Cont.



- Reductions in cache misses and TLB misses follow similar patterns to speedup, suggesting that more efficient memory access is causing our performance gains.

Conclusions and Future Work

- Our storage format has an average of **2.7x** compression compared to Sparsity
- Our method achieved an average of **2.7x** speedup
- Dynamic blocking and reordered block computations significantly reduce cache and TLB misses.
- Pre-processing time is always within **5x** of Sparsity

Future work included demonstrating the compression and speedup by our method are robust over a large set of sparse matrices and identifying the classes for which they are the most beneficial. Furthermore, the pre-processing heuristics can be improve and further evaluated. Lastly, we have not considered optimizations via parallelism, which these methods are very amenable to.

Bibliography

Im, Eun-Jin, and Katherine Yelick. "Optimizing sparse matrix computations for register reuse in SPARSITY." Computational Science—ICCS 2001. Springer Berlin Heidelberg, 2001. 127-136.

Nishtala, Rajesh, et al. "When cache blocking of sparse matrix vector multiply works and why." Applicable Algebra in Engineering, Communication and Computing 18.3 (2007): 297-311.

Note, FLAME Working, Kazushige Goto, and Robert van de Geijn. "On reducing TLB misses in matrix multiplication." (2002).

Williams, Samuel, et al. "Optimization of sparse matrix–vector multiplication on emerging multicore platforms." Parallel Computing 35.3 (2009): 178-194.

Toledo, Sivan. "Improving the memory-system performance of sparse-matrix vector multiplication." IBM Journal of research and development 41.6 (1997): 711-725.

Pinar, Ali, and Michael T. Heath. "Improving performance of sparse matrix-vector multiplication." Proceedings of the 1999 ACM/IEEE conference on Supercomputing. ACM, 1999.