A Cache-Oblivious Sparse Matrix–Vector Multiplication Scheme Based on the Hilbert Curve

Albert-Jan N. Yzelman and Rob H. Bisseling

Abstract The sparse matrix–vector (SpMV) multiplication is an important kernel in many applications. When the sparse matrix used is unstructured, however, standard SpMV multiplication implementations typically are inefficient in terms of cache usage, sometimes working at only a fraction of peak performance. Cache-aware algorithms take information on specifics of the cache architecture as a parameter to derive an efficient SpMV multiply. In contrast, cache-oblivious algorithms strive to obtain efficiency regardless of cache specifics. In earlier work in this latter area, Haase et al. (2007) use the Hilbert curve to order nonzeroes in the sparse matrix. They obtain speedup mainly when multiplying against multiple (up to eight) right-hand sides simultaneously.

We improve on this by introducing a new datastructure, called Bi-directional Incremental Compressed Row Storage (BICRS). Using this datastructure to store the nonzeroes in Hilbert order, speedups of up to a factor two are attained for the SpMV multiplication $y = Ax$ on sufficiently large, unstructured matrices.

1 Introduction

Given an $m \times n$ sparse matrix A and a dense vector x, we consider the sparse matrix–vector (SpMV) multiply $y = Ax$, with y a dense result vector. A standard way of storing a sparse matrix A is the Compressed Row Storage (CRS) format [1], which stores data in a row-by-row fashion using three arrays: j, v, and r. The first two arrays are of size $\mathrm{nz}(A)$, with $\mathrm{nz}(A)$ the number of nonzeroes in A, whereas r is of length $m+1$. The array j stores the column index of each nonzero in A, and v stores the corresponding numerical values. The ranges $[r_i, r_{i+1})$ in those arrays correspond

A.-J.N. Yzelman (✉) · R.H. Bisseling
Utrecht University, 3508 TA Utrecht, The Netherlands
e-mail: ALBERT-JAN.YZELMAN@CS.KULEUVEU.BE; R.H.Bisseling@uu.nl

Algorithm 3 SpMV multiplication algorithm calculating $y = Ax$ using CRS

```
for $i = 0$ to $m - 1$ do
    for $k = r[i]$ to $r[i + 1] - 1$ do
        $y[i] = y[i] + v[k] \cdot x[j[k]]$
    end for
end for
```

to the nonzeroes in the ith row of A. A standard SpMV multiply algorithm using CRS is given in Algorithm 3. It writes to y sequentially, and thus performs optimally (regarding y) in terms of cache efficiency. Accesses to x, however, are unpredictable in case of unstructured A, causing cache misses to occur on its elements. This is the main reason for inefficiencies during the SpMV multiply [3, 5, 6, 12].

A way to increase performance is to force the SpMV multiply to work only on smaller and uninterrupted subranges of x, such that the vector components involved fit into cache. This can be done by permuting rows and columns from the input matrix so that the resulting structure forces this behaviour when using standard CRS. Results on this method have been reported in [16], using a one-dimensional (1D) method, and in [17], where the method has been extended to two dimensions (2D). It must be noted that the 2D method theoretically requires a different datastructure than CRS, but results show that CRS can still outperform more complex datastructures when an appropriate permutation strategy is used. Gains can be as large as 50% for the 1D method and 63% for the 2D method.

What we consider in this paper is a change of datastructure instead of a change in the input matrix structure. This means finding a datastructure which accesses nonzeroes in A in a more "local" manner; that is, an order such that jumps in the input and output vector remain small and thus yield fewer cache misses. Earlier work in this direction includes the Blocked CRS format [11], the auto-tuning sparse BLAS library OSKI [13], exploiting variable sized blocking [10, 14], and several other approaches [2, 12]. In the dense case, relevant are the work by Goto et al. [4], who hand-tuned dense kernels to various different architectures, and the ATLAS project [15], which strives to do the same using auto-tuning. Of specific interest is the use of space-filling curves to improve cache locality in the dense case, in particular the use of the Morton (Z-curve) ordering [9], more recently combined with regular row-major formats to form hybrid-Morton formats [8].

In the sparse case, the work by Haase et al. in [5], which already contains the foundation of the main idea presented here, is of specific interest. They propose to store the matrix in an order defined by the Hilbert curve, making use of the good locality-preserving attributes of this space-filling curve. Figure 1 shows an example of a Hilbert curve within 2×2 and 4×4 matrices. This locality means that, from the cache perspective, accesses to the input and output vector remain close to each other when following the Hilbert curve. The curve is defined recursively as can be seen in the figure: any one of the four "super"-squares in the two-by-two matrix can readily be subdivided into four subsquares, onto which a rotated version of the original curve is projected such that the starting point is on a subsquare adjacent to

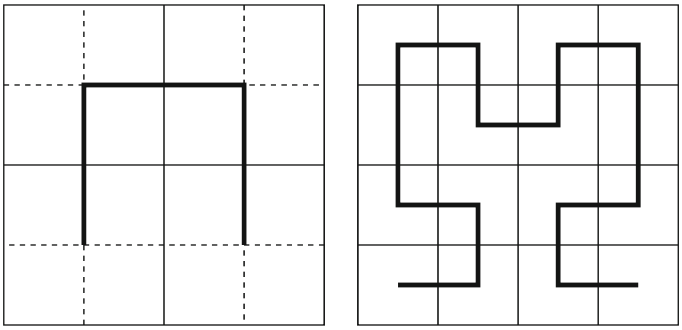

Fig. 1 The Hilbert curve drawn within two-by-two and four-by-four matrices

where the original curve entered the super-square, and similarly for the end point. A Hilbert curve thus can be projected on any $2^{\lceil \log_2 m \rceil} \times 2^{\lceil \log_2 n \rceil}$ matrix, which in turn can embed the sparse matrix A, imposing a 1D ordering on its nonzeroes. Haase et al. [5] stored these nonzeroes in *triplet format*: three arrays i, j, v of length $\mathrm{nz}(A)$ are defined, such that the kth nonzero of A with value $v[k]$ is stored at the location $(i[k], j[k])$, as determined by the Hilbert ordering. The main drawback is the difference in storage space required; this is $3 \cdot \mathrm{nz}(A)$, an increase of $\mathrm{nz}(A) - m$ compared to the standard CRS datastructure. The number of cache misses prevented thus must overtake this amount of extra data movement before any gain in efficiency becomes noticeable.

A new datastructure is proposed in Sect. 2 to alleviate this problem, and results of experiments using the Hilbert curve and this new data format are presented in Sect. 3. These are followed by the conclusions in Sect. 4.

2 Bi-directional Incremental CRS

If using the Hilbert curve to store the nonzeroes of a sparse matrix can be said to be the first of two main ideas around this cache-oblivious method, the second enabling idea is the Bi-directional Incremental CRS datastructure (BICRS). It is capable of efficiently storing the nonzeroes of A in the Hilbert order. We will introduce BICRS by deriving it from the Incremental CRS datastructure (ICRS), which can be viewed as an alternative implementation of the standard CRS datastructure, as presented by Koster [7]. Instead of storing the j array, an incremental version Δj is stored instead; that is, $\Delta j[0] = j[0]$ and $\Delta j[q] = j[q] - j[q-1]$ for $1 \leq q < n$. This means that an SpMV multiplication kernel, upon processing the kth nonzero, simply increases its current column index with $\Delta j[k]$ to find the column index of the next nonzero to be processed. A row change can be signalled by overflowing the column index such that subtracting n from the overflowed index yields the starting column index on the next row. The row array r can then be exchanged for an incremental row array Δr as well, so that $\Delta r[k]$ yields the distance between the kth nonempty row and the next nonempty row. $\Delta r[0]$ specifies which row contains the first nonzero. Note that when there are no empty rows, Δr contains only 1-values except at $\Delta r[0]$, which equals 0. This means the array does not have to be stored, bringing the total storage requirement down to $2{\cdot}\mathrm{nz}(A)$. When the row increment array is stored, the storage requirement is equal to that of CRS with $2{\cdot}\mathrm{nz}(A) + m$, worst case; in the case where A has empty rows, the required storage is less. The main gain is that the SpMV multiply can be efficiently written using pointer arithmetic, which yields a decrease in machine code instructions [7].

As described, ICRS is not capable of storing nonzeroes in any ordering other than the CRS ordering. A simple extension, however, is to allow negative increments, thus facilitating jumping through nonzeroes of the sparse matrix in any bi-directional, possibly non-CRS, order. Overflows in the column-direction still trigger row changes, as with ICRS. We refer to this generalised datastructure as Bi-directional ICRS. An immediate disadvantage is that the row increments array now can become larger than the number of nonempty rows if nonzeroes are not traversed in a row-by-row manner. This hampers efficiency since the number of memory accesses required to traverse A increases to $2{\cdot}\mathrm{nz}(A) + m_{\text{jumps}}$, where m_{jumps} is the number of row jumps stored in Δr, with $m \leq m_{\text{jumps}} \leq \mathrm{nz}(A)$. It is, however, a definite improvement over the triplet structure used in [5]. In case of a dense matrix, the number of row jumps made when nonzeroes are ordered according to the Hilbert curve is about $\mathrm{nz}(A)/2$, but this gives no guarantee for the number of jumps in the sparse case; this is entirely dependent on the nonzero structure. Note that while this datastructure is bi-directional, the datastructure orientation still matters.

3 Experiments

Experiments have been performed on two quad-core architectures, using only one of the four cores available for the sequential SpMV multiplications. The first is an Intel Core 2 Q6600 with a 32 KB L1 data cache, and a 4 MB semi-shared L2 cache. No L3 cache is available. The second architecture is an AMD Phenom II 945e on which each core has a private 64 KB L1 and 512 kB L2 data cache, while all four cores together share a 6 MB L3 cache. The SpMV kernels,[1] based on CRS, ICRS and BICRS using Hilbert ordering, are each executed 100 times on given matrices, and report an average running time of a single SpMV multiplication. Experiments have been performed on 9 sparse matrices, all taken to be large in the sense that the input and output vector do not fit into the L2 cache; see Table 1(top). All matrices are available through the University of Florida sparse matrix collection. Tests on smaller matrices were performed as well, but, in contrast to when using the reordering methods, any decrease in L1-cache misses did not

[1]The source code is freely available at http://albert-jan.yzelman.net/software

Table 1 Matrices used in our experiments (top) and SpMV timings (bottom). An S (U) indicates that a matrix is considered structured (unstructured). Experiments were done on the Intel Q6600 (bottom-left) and the AMD 945e (bottom-right). Timings are in milliseconds

Name	Rows	Columns	Nonzeroes		Symmetry, origin
Stanford	281903	281903	2312497	U	Link matrix
cont1_l	1918399	1921596	7031999	S	Linear programming
Stanford-berkeley	683446	683446	7583376	U	Link matrix
Freescale1	3428755	3428755	17052626	S	Circuit design
Wikipedia-20051105	1634989	1634989	19753078	U	Link matrix
cage14	1505785	1505785	27130349	S	Struct. symm., DNA
GL7d18	1955309	1548650	35590540	U	Combinatorial problem
Wikipedia-20060925	2983494	2983494	37269096	U	Link matrix
Wikipedia-20070206	3566907	3566907	45030389	U	Link matrix

		Hilbert	Extra					Hilbert	Extra
CRS	ICRS	BICRS	build	Matrix		CRS	ICRS	BICRS	build
30.22	40.24	25.74	1456	Stanford	U	22.15	27.52	18.48	832
44.02	46.41	62.85	5085	cont1_l	S	31.07	26.99	48.05	3084
35.29	34.56	45.82	5578	Stanford-berkeley	U	26.05	24.52	34.29	3415
122.27	131.52	210.10	14458	Freescale1	S	98.55	95.00	148.04	8913
366.45	374.82	253.45	12632	Wikipedia-20051105	U	368.36	387.39	250.30	5850
136.19	141.07	165.21	20453	cage14	S	116.44	110.69	140.20	12095
774.55	856.16	372.25	22126	GL7d18	U	716.32	824.32	452.89	10064
812.42	831.17	576.67	23839	Wikipedia-20060925	U	823.53	879.53	550.00	11814
1012.73	994.35	776.48	27345	Wikipedia-20070206	U	1033.95	1124.02	591.08	14753

result in a faster SpMV execution. Results on larger matrices in terms of wall-clock time are reported in Table 1 for the Q6600 system (bottom-left), as well as for the AMD 945e (bottom-right). Also reported is the extra build time, that is, the time required to build the Hilbert BICRS structure minus the time required to build a CRS datastructure.

4 Conclusions

The cache-oblivious SpMV multiplication scheme works very well on large unstructured matrices. In the best case (the GL7d18 matrix on the Q6600), it gains 51% in execution speed. On both architectures, 5 out of the 6 unstructured matrices show significant gains, typically around 30%–40%. The only exception is the stanford-berkeley matrix, taking a performance hit of 32%, on both architectures. Interestingly, the 1D and 2D reordering methods also do not perform well on this matrix [16, 17]. The method also shows excellent performance regarding pre-processing times, taking a maximum of 28 s for wikipedia-2007 on the Q6600 system. This is in contrast to 1D and 2D reordering methods, where pre-processing times can take hours for larger matrices, e.g., 21 h for wikipedia-2006 [16, 17].

Gains in efficiency when reordering, however, are more pronounced than for the Hilbert-curve scheme presented here. Note that the methods do not exclude each other: 1D or 2D reordering techniques can be applied before loading the matrix into BICRS using the Hilbert ordering to gain additional efficiency. The results also show that, as expected, the method cannot outperform standard CRS ordering when the matrix already is favourably structured, resulting in slowdowns.

For future improvement of the Hilbert-curve method, we suggest applying the Hilbert ordering to small (e.g., 8 by 8) sparse submatrices of A instead of its nonzeroes, and imposing a regular CRS ordering on the nonzeroes contained within each such submatrix. Such a hybrid scheme has also been suggested for dense matrices [8], although the motivation differs; in our case, since BICRS can still be used, the number of row jumps required is reduced in case the rows of the submatrices contain several nonzeroes, thus increasing performance further.

References

1. Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., van der Vorst, H. (eds.): Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. SIAM, Philadelphia, PA (2000)
2. Bender, M.A., Brodal, G.S., Fagerberg, R., Jacob, R., Vicari, E.: Optimal sparse matrix dense vector multiplication in the I/O-model. In: Proceedings 19th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 61–70. ACM Press, New York (2007)
3. Dennis, J.M., Jessup, E.R.: Applying automated memory analysis to improve iterative algorithms. SIAM J. Sci. Comput. **29**(5), 2210–2223 (2007)
4. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, Department of Computer Sciences (2002)
5. Haase, G., Liebmann, M., Plank, G.: A Hilbert-order multiplication scheme for unstructured sparse matrices. Int. J. Parallel, Emergent Distr. Syst. **22**(4), 213–220 (2007)
6. Im, E.J., Yelick, K.A.: Optimizing sparse matrix computations for register reuse in SPARSITY. In: Proceedings International Conference on Computational Science, Part I, *Lecture Notes in Computer Science*, vol. 2073, pp. 127–136, Springer, Berlin (2001)
7. Koster, J.: Parallel templates for numerical linear algebra, a high-performance computation library. Master's thesis, Utrecht University, Department of Mathematics (2002)
8. Lorton, K.P., Wise, D.S.: Analyzing block locality in Morton-order and Morton-hybrid matrices. SIGARCH Comput. Archit. News **35**(4), 6–12 (2007)
9. Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada (1966)
10. Nishtala, R., Vuduc, R.W., Demmel, J.W., Yelick, K.A.: When cache blocking of sparse matrix vector multiply works and why. Appl. Algebra Engrg. Comm. Comput. **18**(3), 297–311 (2007)
11. Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings Supercomputing 1999, p. 30. ACM Press, New York (1999)
12. Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM J. Res. Dev. **41**(6), 711–725 (1997)
13. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. J. Phys. Conf. Series **16**, 521–530 (2005)
14. Vuduc, R.W., Moon, H.J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In: High Performance Computing and Communications 2005, *Lecture Notes in Computer Science*, vol. 3726, pp. 807–816, Springer, Berlin (2005)

15. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput. **27**(1–2), 3–35 (2001)
16. Yzelman, A.N., Bisseling, R.H.: Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. SIAM J. Sci. Comput. **31**(4), 3128–3154 (2009)
17. Yzelman, A.N., Bisseling, R.H.: Two-dimensional cache-oblivious sparse matrix–vector multiplication, Parallel Comput. **37**(12), 806–819 (2011)