Efficient sparse matrix multiple-vector multiplication using a bitmapped format

Ramaseshan Kannan
School of Mathematics, The University of Manchester,
Manchester M13 9PL, UK
and
Arup | Oasys,
13 Fitzroy Street, London W1T 4BQ UK.
rkannan@maths.manchester.ac.uk

***Abstract*—The problem of obtaining high computational throughput from sparse matrix multiple–vector multiplication routines is considered. Current sparse matrix formats and algorithms have high bandwidth requirements and poor reuse of cache and register loaded entries, which restrict their performance. We propose the mapped blocked row format: a bitmapped sparse matrix format that stores entries as blocks without a fill overhead, thereby offering blocking without additional storage and bandwidth overheads. An efficient algorithm decodes bitmaps using de Bruijn sequences and minimizes the number of conditionals evaluated. Performance is compared with that of popular formats, including vendor implementations of sparse BLAS. Our sparse matrix multiple-vector multiplication algorithm achieves high throughput on all platforms and is implemented using platform neutral optimizations.**

I. Introduction

The sparse matrix $\times$ vector (SpMV) and sparse matrix $\times$ multiple-vector (SMMV) multiplication routines are key kernels in many sparse matrix computations used in numerical linear algebra, including iterative linear solvers and sparse eigenvalue solvers. For example, in the subspace iteration method used for solving for a few eigenvalues of a large sparse matrix A, one forms the Rayleigh quotient (projection) matrix $M = S^T AS$, where $A \in \mathbb{R}^{n\times n}$ and $S \in \mathbb{R}^{n\times p}$ is a dense matrix with $p \ll n$. The computational bottleneck in such algorithms is the formation of the SMMV products. SpMV/SMMV routines typically utilize only a fraction of the processor's peak performance. The reasons for the low utilisation are a) indexing overheads associated with storing and accessing elements of sparse matrices and b) irregular memory accesses leading to low reuse of entries loaded in caches and registers.

Obtaining higher performance from these kernels is an area of active research owing to challenges posed by hardware trends over the last two decades and significant attention has been paid to techniques that address the challenges. This hardware trend is outlined by McKee in the famous note 'The Memory Wall' (1), (2) and can be summed as follows: the amount of computational power available (both the CPU cycle time and the total number of available cores) is increasing with a rate that is much higher than the rate of increase of memory bandwidth. It will therefore lead to a scenario where performance bottlenecks arise not because of processors' speeds but from the rate of transfer of data to them. The implication of this trend is that there is an increasing need for devising algorithms, methods and storage formats that obtain higher processor utilization by reducing communication. Such techniques and approaches will hold the key for achieving good scalability in serial and parallel execution, both on existing and emerging architectures.

In this paper we introduce a blocked sparse format with an accompanying SMMV algorithm that is motivated by the above discussion of reducing communication cost. The format improves on an existing blocked sparse format by retaining its advantages whilst avoiding the drawbacks. An algorithm that computes SMMV products efficiently for a matrix in this format is developed and its performance is compared with the existing blocked and unblocked formats. The algorithm achieves superior performance over these formats on both Intel and AMD based x86-platforms and holds promise for use in a variety of sparse matrix applications. The current discussion is in the context of a desktop–based structural analysis software package.

II. Overview and comparison of Compressed Sparse Row and Block Compressed Sparse Row formats

The Compressed Sparse Row (CSR) format (3) (or its variant, the Compressed Sparse Column format) can be regarded as the *de-facto* standard format for storing and manipulating sparse matrices. The CSR format stores the nonzero entries in an array `val` of the relevant datatype (single precision, double precision or integers). The column indices of the entries are stored in `col_idx` and the row indices are inferred from the markers into `col_idx`, stored as `row_start`. For example, with array indices starting with 0:

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{pmatrix}$$

$$\begin{aligned} \texttt{val} &= (a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{22}, a_{23}, a_{32}) \\ \texttt{col\_idx} &= (0, 1, 2, 3, 0, 1, 2, 3, 2) \\ \texttt{row\_start} &= (0, 4, 6, 8, 9) \end{aligned}$$

```
1  for i = 0 to n − 1
2    yi = y[i]
3    for j = row_start[i] to row_start[i + 1]
4      yi += val[j] * x[col_idx[j]]
5    y[i] = yi
```

Fig. 1. Compute $y = y + Ax$ for a matrix A stored in CSR format and confirming vectors x and y stored as arrays.

If the number of nonzeros in A is z and if A is stored in double precision, the storage cost for A in the CSR format is $3z + n + 1$ words. (We assume a word size of 32 bits for the entire discussion.) An unoptimized SpMV algorithm is shown in snippet 1.

The SpMV implementation in Algorithm 1 suffers from the problem of irregular memory use, which results in reduced data locality and poor reuse of entries loaded in registers. It performs a single floating point addition and multiplication for every entry `val` and `x` loaded, thus has a low 'computational intensity', i.e., it performs too few flops for every word of data loaded. This aspect of CSR SpMV has been well studied in the past, see (4) or (5) for example.

The Block Compressed Sparse Row (BCSR) format (5) is intended to improve the register reuse of the CSR. The BCSR format stores nonzero entries as dense blocks in a contiguous array `val`. These blocks are of size $r \times c$ where r and c are respectively the number of rows and columns in the dense blocks. For indexing it stores the column position of the blocks in an array (`col_idx`) and row-start positions in `col_idx` in `row_start`.

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{pmatrix}$$

$$\begin{aligned} \mathtt{r} &= 2, \mathtt{c} = 2 \\ \mathtt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, \\ &\quad a_{22}, a_{23}, a_{32}, 0) \\ \mathtt{col\_idx} &= (0, 1, 1) \\ \mathtt{row\_start} &= (0, 2, 3) \end{aligned}$$

A sparse matrix stored in the BCSR format takes up $\approxeq 2brc + b + \frac{n}{r}$ words to store, where b is the number of nonzero blocks (for a given r and c) stored when the matrix is held in BCSR.

The SpMV routine for BCSR with $r, c = 2$ is presented in Algorithm 2. Since the `val` array is stored as a sequence of blocks, the algorithm loads all entries in a block into registers and multiplies them with corresponding entries in `x`. The increase in spatial locality results in the reuse of register–loaded entries of `x`, reducing the total number of cache accesses. The inner loop that forms the product of the block with the corresponding part of the vector is fully unrolled, reducing branch penalties and allowing the processor to prefetch data. There is, however, a tradeoff involved in selecting the right block size for BCSR. The reuse of loaded registers increases in number with an increase in r and c

```
1  for i = 1 to bm
2    ir = i × r
3    y0 = y[ir]
4    y1 = y[ir + 1]
5    for j = row_start[i] to row_start[i + 1]
6      jc = col_idx[j] × c
7      y0 += val[0] * x[jc]
8      y1 += val[2] * x[jc]
9      y0 += val[1] * x[jc + 1]
10     y1 += val[3] * x[jc + 1]
11     increment pointer val by 4
12   y[ir] = y0
13   y[ir + 1] = y1
```

Fig. 2. Compute $y = y + Ax$ for a matrix A in BCSR with $r, c = 2$ and bm block-rows and vectors x, y stored as arrays.

but having larger block sizes may increase the fill, leading to higher bandwidth costs and extra computations involving zeros, which decrease performance. The amount of fill for a given block size depends on the distribution of the nonzeros in the matrix. The efficiency gains from increasing the block size will depend on the size of the registers and the cache hierarchy of the machine the code is running on. There is, therefore, an optimal block size for a given matrix (or set of matrices with the same nonzero structure) and a given architecture. This suggests using a tuning based approach to picking the optimum block size and such approaches have been studied extensively in (6), (7) and (8). The dependence of the performance of the SpMV routine on the structure of the matrix make it a complex and tedious process to extract enough performance gains to offset the overheads of maintaining and manipulating the matrix in a different format, such as implementing kernels for other common matrix operations for a given block size. Additionally, the prospect of storing zeros increases the storage costs, making it dependent on the matrix structure, which is information that is available only at runtime. The arguments above motivate a blocked format that offers the benefits of BCSR's performance without the associated storage and bandwidth penalties.

III. The Mapped Blocked Row format

We now introduce the mapped blocked row sparse (MBR) format for storing sparse matrices. For a matrix

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & a_{32} & 0 \end{pmatrix},$$

we represent the 2×2 block on top right as a combination of the nonzero elements and a boolean matrix representing the nonzero structure:

$$\begin{matrix} a_{02} & a_{03} \\ 0 & 0 \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_{02} & a_{03} \end{matrix} + \begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix}.$$

The bit sequence `0011` can be represented in decimal as 3 and this representation is stored in a separate array. Thus, for our

example, the MBR representation can be written as follows:

$$\begin{aligned} \texttt{r} &= 2, \texttt{c} = 2 \\ \texttt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \\ \texttt{col\_idx} &= (0, 1, 1) \\ \texttt{b\_map} &= (15, 3, 7) \\ \texttt{row\_start} &= (0, 2, 3) \end{aligned}$$

The bit structures of the blocks are stored in `b_map`, the array of their corresponding decimal representations. The bitmaps are encoded in left-to-right and top-to-bottom order, with first position in the block (i.e. the bit on top left) being stored as the lowest bit and the bottom right position being the highest.

The datatype `maptype` of the `b_map` array can chosen to fit the size of the blocks; hence if the block size is 8×5, 40 bits are required and a `__int64` (9) will be used but if the block size is 4×4, a `maptype` of `short` will suffice for the 16 bits needed. For block sizes where the number of bits are less than the corresponding variable that stores the bitmap, the excess bits are left unused. With built-in C++ datatypes, up to 8×8 blocks can be supported. Larger blocks can be constructed by combining two or more adjacent instances in an array of built-in types or by using a dynamic bitset class like `boost::dynamic_bitset` [1].

The storage cost of an $n \times n$ matrix in the MBR format is bounded by

$$\underbrace{2z}_{\texttt{val}} + \underbrace{b}_{\texttt{col\_idx}} + \overbrace{\frac{b}{\delta}}^{\texttt{b\_map}} + \underbrace{\frac{n}{r}}_{\texttt{row\_start}} \quad \text{words}$$

where z is the number of nonzeros, b the number of blocks and r the size of the blocks. δ is the ratio $\frac{\texttt{sizeof(int)}}{\texttt{sizeof(maptype)}}$ to convert the size of `maptype` into words. b lies in a range that is given by the following lemma.

Lemma III.1. *For an $n \times n$ sparse matrix with z nonzeros and at least one nonzero per row and per column, the minimum number $b_{\min}$ of $r \times r$ blocks required to pack the entries is* $\mathrm{ceil}(z/r^2)$ *and the maximum $b_{\max}$ is* $\min(z, \mathrm{ceil}(n/r)^2)$.

Proof: Since $z > n$, z entries can be arranged such that there is at least one nonzero per row and column. This can be done in z/r^2 blocks but no less, hence $b_{\min} = \mathrm{ceil}(z/r^2)$ is the minimum number of blocks. The $n \times n$ matrix contains $\mathrm{ceil}(\frac{n}{r})^2$ blocks. If $z > \frac{n^2}{r^2}$, then $b_{\max} = \mathrm{ceil}(\frac{n}{r})^2$, otherwise each nonzero can occupy a block of its own, so we have $b_{\max} = z$ blocks. ■

Although these bounds would be seldom attained in practice, they can provide an intuitive feel for when a particular format can become advantageous or disadvantageous. The storage costs of CSR, BCSR and MBR are compared in Table I. For the lower bound of b, the MBR format takes up storage comparable with the CSR format but more than BCSR. For b close to the upper bound, MBR requires significantly less storage than BCSR term but more than CSR. However, for all our test matrices, the number of blocks arising from the conversion to blocked formats resulted in MBR requiring less storage than both BCSR and CSR. Table II lists the storage costs (in words) and their ratios arising for 8×8 blocking of the test matrices (the matrices are introduced in Table III).

[1] http://www.boost.org/doc/libs/1_36_0/libs/dynamic_bitset/dynamic_bitset.html

TABLE I. COMPARISON OF STORAGE BOUNDS FOR CSR, BCSR AND MBR.

	CSR	BCSR	MBR
	$3z + n$	$2br^2 + b + \frac{n}{r}$	$2z + b\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$
Lower bound	$3z + n$	$2z + \frac{z}{r^2} + \frac{n}{r}$	$2z + \frac{z}{r^2}\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$
Upper bound	$3z + n$	$2n^2 + \frac{n^2}{r^2} + \frac{n}{r}$	$2z + \frac{n^2}{r^2}\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$

TABLE II. RATIO OF STORAGE FOR MBR TO BCSR AND MBR TO CSR FORMATS FOR 8×8 BLOCKS.

Matrix	n	b	MBR / BCSR	MBR / CSR
sp_hub	143,460	249,267	0.171	0.759
rajat29	643,994	991,244	0.1	0.839
nlpkkt80	1,062,400	2,451,872	0.205	0.744
hamrle3	1,447,360	906,839	0.119	0.774
ecology1	1,000,000	622,750	0.149	0.75
dielFilterV3	1,102,824	11,352,283	0.145	0.791
dielFilterV2	1,157,456	8,106,718	0.116	0.828
asic_680k	682,862	728,334	0.106	0.814

A. Similarity with other formats

Buluç et al. independently propose a format called the 'bitmasked CSB' (10), based around the idea of storing blocks that are compressed using a bit structure representation. The format partitions the matrix into "compressed sparse" blocks of bitmapped register blocks, resulting in two levels of blocking. The nonzero entries in each register block are stored contiguously and their positions within the block are marked using a bitwise representation. There is no storage of zeros (i.e. the fill), which improves on BCSR in the same way that MBR does, but the CSB SpMV algorithm does perform multiply-add operations on zeros so as to avoid conditionals. The storage cost for MBR is slightly less than that of bitmasked CSB because of higher bookkeeping arising from two levels of blocking and there are subtle differences in the encoding of bit structure of the blocks. In order to perform the multiplication of a block with the relevant chunk of a vector, bitmasked CSB uses SIMD instructions to load matrix entries, which we avoid. Instead, in MBR SpMV, we minimize the conditionals evaluated using de Bruijn sequences. Overall, whilst bitmasked CSB is geared towards optimizing parallel execution, the aim of this work is to obtain a high throughput for multiple vectors in the sequential case.

IV. SPMV AND SMMV WITH MBR

As noted in the first section, the SMMV kernels are employed in sparse eigensolvers, and our interest is in their eventual use in the commercial desktop software for structural analysis Oasys GSA (11). This software is required to run on a variety of x86 architectures both old and new. The aim with implementing SpMV and SMMV for the MBR format, therefore, was to obtain a clean but efficient routine subject to the following considerations.

- Not employing optimizations that are specific to a platform or hardware, for example prefetching.

```
1  for each block row bi
2    for each block column bj in block row bi
3      map = b_map_bj
4      for each bit map_p in map
5        if map_p = 1
6          i:= p%r + bi × r, j:= p%c + bj × c
7          y(i)+ = ⋆val × x(j)
8          increment val
9        end
10     end
11   end
12 end
```

Fig. 3. SpMV for a matrix stored in MBR with block dimensions (r, c) and confirming vectors x, y stored as arrays.

- Obtaining kernels with parameters such as the type of scalar (single, double, higher precision), block sizes, number of dense vectors and `maptype` datatypes bound at compile time. C++ templates offer metaprogramming techniques that allow such kernels to be generated at the time of compilation from a single source base. This approach is advantageous compared with the use of external code generators for generating kernels in a parameter space since the programmer can write code for generating kernels in the same environment as generated code, thus making it easier to maintain and update.
- Focussing on sequential execution; this will inform the approach for a parallel version, which will be tackled as future work.

The programming language used was C++, using templates for kernel generation and to provide abstractions for optimizations like loop unrolling. The compilers used were Microsoft Visual C++ and Intel C++.

A naïve algorithm for SpMV for MBR is shown in Algorithm 3. The notation `⋆val` indicates dereferencing the C pointer `val` to obtain the value of the nonzero entry it currently points to. Let z_b be the number of nonzeros in a given block, represented as set bits in `map`. It is easy to show that the minimum number of register loads required to multiply a block by a corresponding chunk of vector x and add the result to y will be $\mathcal{O}(3z_b)$ in the worst case. Algorithm 3 attains this minimum and also minimizes flops since it enters the block–vector product loop (steps 6–8) exactly z_b times.

However, the algorithm is inefficient when implemented, because steps 4–11 contain conditionals that are evaluated r^2 times, which the compiler cannot optimize. The presence of conditionals also implies a high number of branch mispredictions at runtime since the blocks can have varying sparsity patterns. Mispredicted branches necessitate removal of partially-completed instructions from the CPU's pipeline, resulting in wasted cycles. Furthermore, step 6 is an expensive operation because of modulo (remainder) calculation (denoted using the binary operator `%`) and the loop does not do enough work to amortize it.

We introduce a set of optimizations to overcome these inefficiencies.

```
1  ⋮
2  for each set bit p in map
3    i:= p%r + bi × r, j:= p%c + bj × c
4    y(i)+ = ⋆val × x(j)
5    increment val
```

Fig. 4. A modification to steps 4–8 Algorithm 3.

A. Optimal iteration over blocks

The `for` loop in step 4 and the `if` statement in line 5 contain conditionals (in case of the `for` loop, the conditional is the end-of-loop check) that present challenges to branch predictors. The true/false pattern of the second conditional is the same as bit-pattern of the block, the repeatability of which decreases with increasing size of the block.

One workaround is to use code replication for each bit pattern for a given block size, such that all possible bit structures are covered exhaustively. It would then be possible to write unrolled code for each configuration, thus completely avoiding conditionals. However this quickly becomes impractical since there are 2^r arrangements for a block of size $r \times r$ and generating explicit code can become unmanageable. Additionally, since it is desirable to not restrict the kernel to square blocks, the number of configurations to be covered increases even further. The solution therefore is to minimize the number of conditionals evaluated and fuse the end-of-loop check with the check for the set bit. In other words, instead of looping r^2 times over each block (and evaluating r^2 conditionals), we loop over them z_b times, thus evaluating only z_b conditionals. Algorithm 4 shows the modification to the relevant section from Algorithm 3.

The key detail is iterating over *set* bits in 'for each'. This is achieved by determining the positions of trailing set bits using constant time operations. Once the position is determined, the bit is unset and the process is repeated till all bits are zero. To determine the positions of trailing bits, we first isolate the trailing bit and then use de Bruijn sequences to find its position in the word, based on a technique proposed by Leiserson et al in (12).

A de Bruijn sequence of n bits, where n is a power of 2, is a constant where all contiguous substrings of length $\log_2 n$ are unique. For $n = 8$, such a constant could be $C :=$ `00011101`, which has all substrings of length 3 (`000`, `001`, `011`,..., `101`, `010`, `100`) unique. A lone bit in an 8-bit word, say x, can occupy any position from 0 to 7, which can be expressed in 3 bits. Therefore, by operating x on C, a 3-bit distinct word can be generated. This word is hashed to the corresponding position of `1` in x and such a hash table is stored for all positions, at compile time. At run time, the procedure is repeated for an x with a bit at an unknown position to yield a 3-bit word, which can then be looked up in the hash table.

Algorithm 5 lists the realization of 'for each' and the decoding of map. Step 3 isolates the trailing bit into y using the two's complement of map, steps $4-6$ calculate the index of the bit and step 9 clears the trailing bit. The operators used in the algorithm are standard C/C++ bitwise operation symbols: `&` for bitwise `AND`, `<<` and `>>` for left shift and right shift by the number denoted by the second operand, $\sim$

```
1  Pick an 8 bit de Bruijn sequence C and generate its hashtable h
2  while map ≠ 0
3    y = map & (−map)
4    z = C × y
5    z = z >> (8 − log2 8)
6    p = h(z)
7    compute i and j from p and multiply (Alg. 3 steps 6–8)
8    ⋮
9    map = map & (∼y)
10 end
```

Fig. 5. Looping over set bits for a bitmap x of length 8 bits.

```
1  Given ... and x1 ··· xℓ , y1 ··· yℓ ∈ R^n
2  for each block row bi
3    for each block column bj in row bi
4      map = b_map_bj
5      for each set bit p in map
6        i:= p%r + bi × r, j:= p%c + bj × c
7        y1(i)+ = ⋆val × x1(j)
8          ⋮
9        yℓ(i)+ = ⋆val × xℓ(j)
10       increment val
```

Fig. 6. Multiplying multiple vectors in inner loops.

for bit complement. Steps 3–6 can be carried out in constant time and bitwise operations execute in a constant number of clock cycles (13). The constant time decoding, combined with a reduction in the number of conditionals evaluated gives huge performance gains.

B. Unrolled loops for multiple vectors

At every iteration of the inner loop, Algorithm 4 calculates the position of the nonzero entry in the block and multiply-adds with the source and destination vector. The cost of index-calculation and looping can be amortized by increasing the amount of work done per nonzero decoded. This can be achieved by multiplying multiple vectors per iteration of the loop.

For the kernel to work with any value of ℓ, either the mutiply-add operations would need to be in a loop, which would be inefficient at runtime or we would need to write ℓ versions of the kernel, which can be tedious and expensive to maintain. This is overcome by using templates that generate code for many kernels at compile time, each varying in the number of vectors and each unrolling the innermost loop ℓ times. There is a close relationship between the performance of the kernel, the size of blocks (r, c) and the number of vectors multiplied ℓ. For a given block size, performance increases with increase in the number of vectors in the inner loop, since it increases the reuse of the nonzero values loaded in the registers and on lower levels of cache, till such a point where loading more vector entries displaces the vectors previously loaded, thereby destroying the benefit blocking brings in the first place. This suggests a need for tuning based either on comparative performance of the kernels or on heuristics gathered from the architecture (or indeed, on both). We use the former approach, leaving the investigation of the latter to future work.

V. Numerical experiments

The experimental setup consists of x86-based machines based on Intel and AMD platforms intended to be representative of the target architectures the kernel will eventually run on. The code is compiled using the Intel C++ compiler v12.1 Update 1 on Windows with full optimization turned on. We note however, that this does not apply to the pre-compiled Intel MKL code (used for benchmarking) that takes advantage of vectorization using SIMD instructions available on all our test platforms. The test platforms consist of machines based on AMD Opteron, Intel Xeon and Intel Sandy Bridge processors. The AMD Opteron 6220, belonging to the Bulldozer architecture, is an 8-core processor with a clock speed of 3.0 GHz and 16 MB of shared L3. Each core also has access to 48 KB of L1 cache and 1000 KB of L2 cache. The Intel Harpertown-based Xeon E5450 on the other hand has access to 12 MB of L2 cache, shared between 4 cores on a single processor, each operating at 3 GHz. Each core has 256 KB of L1 cache. Both the Opteron and Xeon support the 128-bit SSE4 instruction set that allows operating on 2 double floating point number in a single instruction. The third test platform is the Intel Core i7 2600 processor, based on the recent Sandy Bridge architecture, which is the second generation in the Intel Core line of CPUs. This processor has 4 cores sharing 8 MB of shared L3 cache with two levels private caches of 32 KB and 256 KB for each core. The cores operate at a peak clock speed of 3.8 GHz, with Turbo Boost turned off. The Core i7 processor uses the AVX instruction set, supporting 256-bit wide registers that enable operating on 4 double precision variables in a single instruction.

The test matrices consist of a set of matrices from the University of Florida Sparse Matrix collection (14) as well as from problems solved in Oasys GSA. These are listed in Table III.

The final algorithm for MBR SMMV was a combination of all optimizations described in the previous section. The C++ implementation of this was run on the matrices via a test harness for different values of block sizes upto a maximum of 8×8 and multiple vectors. Where matrix sizes are not multiples of the block size, the matrix is padded with zeros on the right and on the bottom, such that the increased dimensions are a multiple. This merely involves modifying the arrays storing the indices and does not affect the storage or the performance, since the blocks are sparse. The performance of the implementation was compared with that of CSR SMMV and BCSR SMMV. For all our tests, a near-exclusive access is simulated by ensuring that the test harness is the only data-intensive, user-driven program running on the system during the course of benchmarking.

We do not study the effect of reordering on the performance of the kernels, since in applications, the specific choice of the reordering algorithm may not always be governed by SMMV, instead it could be governed by other operations that the application performs. We do however note that any reordering approach that decreases the bandwidth of a matrix[2] will, in general, increase performance of a blocked format. Furthermore, we do not consider the costs of conversion between

[2]In this context, the term bandwidth refers to the maximum distance of a matrix nonzero element from the diagonal.

TABLE III. TEST MATRICES USED FOR BENCHMARKING SMMV ALGORITHMS.

	Matrix	Source	Dimension	Nonzeros	Application
1	ASIC_680k	U.Florida	682,862	3,871,773	Circuit simulation
2	atmosmodm	U.Florida	1,489,752	10,319,760	Atmospheric modeling
3	circuit5M	U.Florida	5,558,326	59,524,291	Circuit simulation
4	dielfilterV2real	U.Florida	1,157,456	48,538,952	Electromagnetics
5	dielfilterV3real	U.Florida	1,102,824	89,306,020	Electromagnetics
6	ecology1	U.Florida	1,000,000	4,996,000	Landscape ecology
7	G3_circuit	U.Florida	1,585,478	7,660,826	Circuit simulation
8	hamrle3	U.Florida	1,447,360	5,514,242	Circuit simulation
9	nlpkkt80	U.Florida	1,062,400	28,704,672	Optimization
10	rajat29	U.Florida	643,994	4,866,270	Circuit simulation
11	sp_hub	Arup	143,460	2,365,036	Structural engineering
12	watercube	Arup	68,598	1,439,940	Structural engineering

various formats since applications can generate a sparse matrix in the MBR format and use the format for an entire application, thereby negating expensive data transformations. This does, however, necessitate the availability of software for matrix manipulations and factorizations that the application uses.

In the case of CSR, a standard SpMV implementation based on Algorithm 1 and the functions available from the Intel MKL library (15) are used for comparison. The sparse BLAS Level 2 and Level 3 routines available within the MKL library are regarded as highly optimized implementations and achieve performance higher than corresponding reference implementations, especially on Intel platforms. The library offers the functions `mkl_cspblas_dcsrgemv` and `mkl_dcsrmm` that perform SpMV and SMMV operations. Since our objective is to obtain benchmarks for SMMV, `mkl_dcsrmm` would appear to be the right candidate. However, in almost all our experiments, `mkl_cspblas_dcsrgemv` outperformed `mkl_dcsrmm`, hence `mkl_cspblas_dcsrgemv` was used as the benchmark. Since the MKL library is closed-source software, it is not possible to determine why `mkl_dcsrmm` is not optimized to take advantage of multiple vectors.

For BCSR SMMV, an implementation of Algorithm 2 that is optimized for multiple vectors is used. This uses unrolled loops and multiplies each block with multiple vectors. The right number of vectors to be multiplied within each loop depends on the architecture and has been studied in (8). Similar to the MBR algorithm, the optimal number of vectors depends on the architecture and needs to be selected via tuning. For the purpose of this comparison, we run BCSR SMMV kernels with fixed blocks of sizes 4×4 and 8×8, with increasing number of multiple vectors, going from 1 to 20, and select the best performance as being the representative performance of the format.

A. Performance against number of vectors

The performance of the MBR format depends on amortizing the cost from decoding the blocks, and this is achieved by multiplying multiple vectors. Therefore it is important to know the behaviour of the algorithm with respect to varying ℓ, the number of vectors. Figures 7 and 8 present how the performance varies on the Core i7 and Opteron respectively. In both cases, there is a sharp increase in performance initially, followed by a plateau and then a drop as the implementations go from single-vector kernels to ones handling 20 vectors. The reason for this behaviour is that when the number of vectors is increased, more vector entries stay loaded in the cache, reducing misses when the algorithm tries to load them again. The performance peaks and starts decreasing when the number of vectors reaches a point where loading more entries into the cache displaces previously loaded entries, leading to increased misses. The performance peaks for a different number of vectors, depending on the size of the cache hierarchy and to a lesser extent, matrix sizes and sparsity patterns.

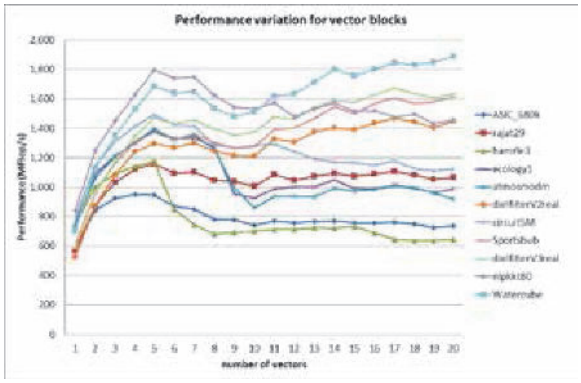

Fig. 7. Performance variation of MBR SMMV across multiple vectors for test matrices on Intel Core i7 2600

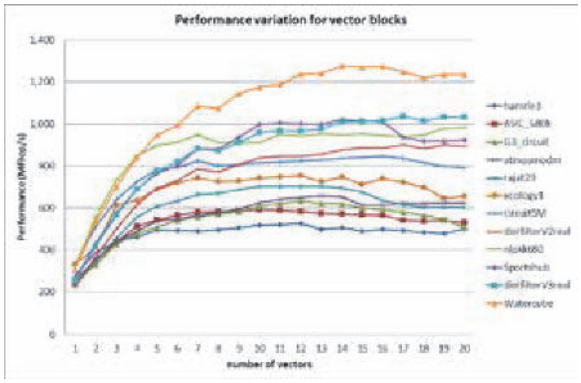

Fig. 8. Performance variation of MBR SMMV across multiple vectors for test matrices on AMD Opteron 6220

On the Opteron, most matrices exhibit peak performance around the range of 12 to 16 vectors whilst on i7, the range is

around 5 to 6. Both processors have a comparable L1 cache size but the Opteron has almost four times L2 cache as Core i7. This allows for more reuse of loaded entries and hence the performance tops at a higher value of ℓ.

A small number of matrices in graph 7 show an increase in performance after hitting a trough in the post-peak part of their curves. These matrices are watercube, sp_hub, dielfilterV2real and dielfilterV3real, from structural engineering and circuit simulation applications. They have the highest nonzero density amongst all matrices but are within the lower half when arranged by increasing order of matrix sizes. This combination of smaller sizes and low sparsity could result in higher performance—the size ensures that a larger number of vector chunks or entire vectors are resident in L3 caches, whereas the higher density results in higher flops per loaded vector entry. Indeed, the best performance is attained for watercube on both processors, which is the smallest matrix but has the highest nonzero density.

B. Performance comparison with other kernels

The performance of the kernels is compared with that of other SMMV routines and the results for different platforms are presented in Figures 9, 10 and 11 for Xeon, Core i7 and Opteron respectively. In the case of MBR and BSR, the kernels used are the ones that yield the maximum performance for the given matrix and for the given block size, i.e. the peaks of graphs for each matrix in figures 7 or 8.

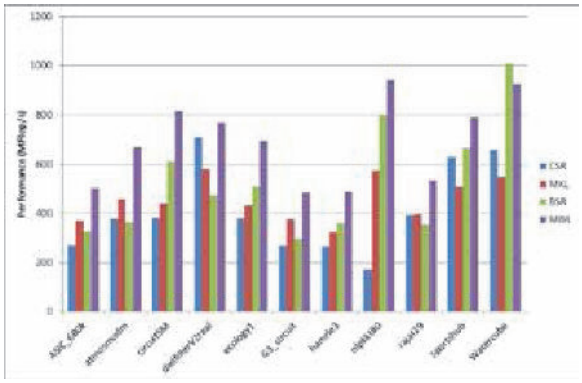

Fig. 9. Performance comparison of MBR SMMV on Intel Xeon E5450

On the Xeon, MBR is faster than MKL by factors of 1.3 to 1.9. It is also more efficient than BCSR for all matrices except watercube. The Xeon has the largest L2 cache of all test platforms. The large L2 cache and the small size of the matrix ensures that BCSR is faster, since it has fully unrolled loops with no conditionals, thus ensuring very regular data access patterns that aid prefetching. Furthermore, watercube also has the highest nonzero density and highest number of entries-per-block (z/b from Table II) so the ratio of redundant flops (i.e. operations involving 0 entries) to useful flops is low, which helps the BCSR routine.

The performance trends for Core i7 are somewhat similar to those of Xeon, but a key difference is that it is the only architecture where MKL outperforms MBR for some matrices. The MKL to MBR performance ratios vary from

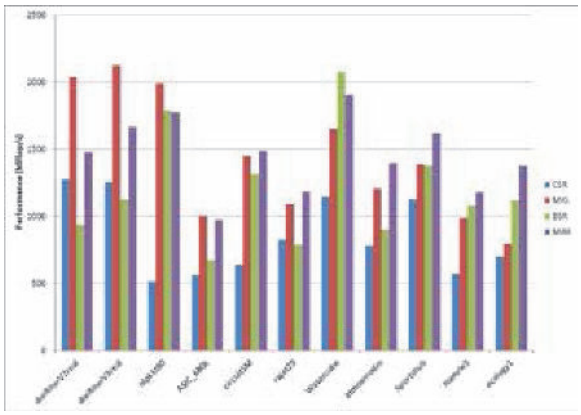

Fig. 10. Performance comparison of MBR SMMV on Intel Core i7 2600

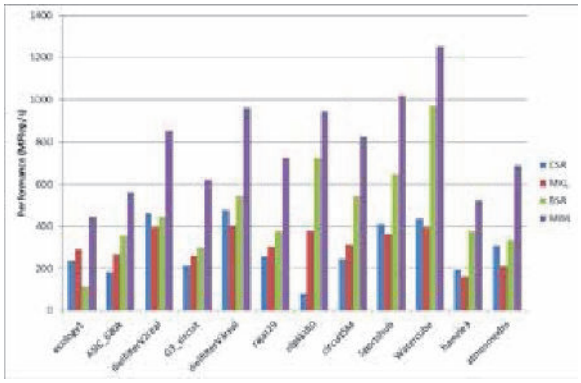

Fig. 11. Performance comparison of MBR SMMV on AMD Opteron 6220

0.72 to 1.71. MKL is faster than MBR on four matrices: dielfilterV2real, dielfilterV3real, ASIC_680k and nlpkkt80, which come from different applications. There are no specific properties of these matrices or their sparsity patterns that gives us a suitable explanation for why MBR is slower. For two of the four matrices–dielfilterV2real and dielfilterV3real–the MBR SMMV performance vs. number of vectors graph (Figure 7) indicates that higher performance could be gained by using more than 20 vectors, although such a kernel may not always be relevant, especially in applications with small number of right hand sides. Evidently, MKL's use of AVX instructions on Sandy Bridge allows for good efficiency gains that lead to a higher throughput. It will need a closer evaluation using experimental data from hardware counters combined with performance modelling to explain the reasons for this discrepancy, which will be looked at in future work.

Finally, on the AMD processor, MBR outperforms MKL by factors of 1.5 to 3.2 and BCSR by factors of 1.3 to 3.9. This demonstrates that while the MKL routines use architecture–specific and platform–specific optimization to gain efficiency, MBR SMMV is capable of attaining high efficiency through platform–neutral optimizations that deliver a good performance on all platforms.

VI. Conclusions and future work

This work introduces mapped blocked row as a practical blocked sparse format that can be used with sparse matrix programs. The storage requirements of the format have been studied and they are significantly less than the two popular formats we have compared with. The MBR format offers the distinct advantage of being a blocked format that does not incur the computational and storage overheads of other formats. This holds promise for applications that involve very large problem sizes where holding the matrix in memory is an issue, for example iterative solvers for linear systems. A C++ implementation of the algorithm offers compile–time parameters like the block size, number of vectors and the datatypes of the scalars and of the bitmap, making it generic in scope for a wide range of applications.

A fast algorithm has been developed for multiplying sparse matrices in the MBR format, with several optimizations for minimizing loop traversals and evaluations of conditionals, for increasing cache reuse and to amortize the decoding costs. By virtue of operating on a blocked format, the algorithm obtains high computational intensity. A C++ implementation of the algorithm offers compile–time parameters like the block size, number of vectors and the datatypes of the scalars and of the bitmap, making it generic in scope for a wide range of applications. The templates also makes it possible to produce code that has fully unrolled loops and kernels that bind to parameters at compile-time, unifying the code generator with the generated code for greater transparency and maintainability.

The performance results presented in the previous section prove that these performance optimizations can achieve good efficiency gains on all platforms by increasing register and cache reuse. The reference implementation attains performance over $3\times$ that of the Intel MKL libraries and better performance on most test platforms over existing optimized BCSR and CSR implementations. There is ample scope to tune performance by modifying parameters such as the block size and effects of such tuning will be the topic of future work. A key motivation for communication reducing algorithms is the desire for improved parallel scalability. This article has focussed on establishing the performance of the MBR format and the algorithm for sequential execution, paving the way for its parallelization, which will be explored in future work. Also of interest is the question "to what extent is a blocked sparse format of relevance for sparse direct solutions?" and whether it can offer advantages for storing and manipulating the factors from a Cholesky or symmetric indefinite factorization. These will be examined in due course.

Acknowledgements

The author would like to thank Professors Nicholas J. Higham and Françoise Tisseur for their comments and feedback. This research was funded by a scholarship from the University of Manchester and a studentship from Arup.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, March 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[2] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, ser. CF '04. New York, NY, USA: ACM, 2004, pp. 162–167. [Online]. Available: http://doi.acm.org/10.1145/977091.977115

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1994.

[4] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 711–726, March 1997.

[5] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/331532.331562

[6] E.-J. Im and K. A. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," in *Proceedings of the International Conference on Computational Science*, ser. LNCS, vol. 2073. San Francisco, CA: Springer, May 2001, pp. 127–136.

[7] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[8] B. C. Lee, R. W. Vuduc, J. W. Demmel, K. A. Yelick, M. deLorimier, and L. Zhong, "Performance optimizations and bounds for sparse symmetric matrix-multiple vector multiply," University of California, Berkeley, CA, USA, Tech. Rep. UCB/CSD-03-1297, November 2003.

[9] *Visual C++ Language reference, available from http://msdn.microsoft.com/en-us/library/3bstk3k5.aspx*, Microsoft Corporation, 2011, retrieved on January 26, 2012.

[10] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE International, May 2011, pp. 721–723.

[11] Oasys Software, "Oasys GSA suite," http://www.oasys-software.com/gsa, Oasys Software, 2011, retrieved on January 20, 2012. [Online]. Available: http://www.oasys-software.com/gsa

[12] C. E. Leiserson, H. Prokop, and K. H. Randall, "Using de Bruijn Sequences to Index a 1 in a Computer World," MIT, Tech. Rep., July 1998, unpublished manuscript, available from

http://supertech.csail.mit.edu/papers/debruijn.pdf. [Online]. Available: http://supertech.csail.mit.edu/papers/debruijn.pdf

[13] A. Fog, "Optimizing software in C++," http://www.agner.org/optimize/optimizing_cpp.pdf, Feb 2012, retrieved on August 07, 2012. [Online]. Available: http://www.agner.org/optimize/optimizing_cpp.pdf

[14] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[15] Intel Corporation, "Intel Math Kernel Library (Intel MKL)," http://software.intel.com/en-us/articles/intel-mkl/, 2012, version 10.3 Update 7.