Generalised vectorisation for sparse matrix–vector multiplication

A. N. Yzelman*†‡
*Flanders ExaScience Lab, Belgium
†Department of Computer Science, KU Leuven
‡Huawei Technologies, France
20, Quai du Point du Jour
92100 Boulogne-Billancourt, France
albertjan.yzelman@huawei.com

***Abstract*—This work generalises the various ways in which a sparse matrix–vector (SpMV) multiplication can be vectorised. It arrives at a novel data structure that generalises three earlier well-known data structures for sparse computations: the Blocked CRS format, the (sliced) ELLPACK format, and segmented scan based formats.**

The new data structure is relevant for sparse computations on modern architectures, since efficient use of new hardware requires the use of increasingly wide vector registers. Normally, the use of vectorisation for sparse computations is limited due to bandwidth constraints. In cases where computations are limited by memory latencies instead of memory bandwidth, however, vectorisation can still help performance. The Intel Xeon Phi, appearing as a component in several top-500 supercomputers, displays exactly this behaviour for SpMV multiplication. On this architecture, the use of the new generalised vectorisation scheme increases performance up to 178 percent.

***Index Terms*—sparse matrix–vector multiplication, cache-oblivious algorithms, shared-memory parallelism, vectorisation, Intel Xeon Phi, Compressed Row Storage, ELLPACK, segmented reduce**

I. Introduction

Sparse matrix–vector (SpMV) multiplication often arises as a key computational component. Its performance is critical for the iterative solution of linear systems and eigensystems, for example, which makes the SpMV a critical kernel for structural engineering, fluid dynamics, graph analysis, chip design, chemistry, geomechanics, optimisation, language processing, visualisation, and other applications. Outside of iterative solvers, another example of a major application area of SpMV multiplications are graph computations.

Modern architectures have increasingly many threads per core and increasingly many cores per processor, while the effectively available bandwidth per core has been declining. This forms a bottleneck for computations where few operations per data word are executed. For computations like those, CPUs cannot reach peak performance because they have to wait for data to operate on, making the algorithm execution time bounded by CPU-to-RAM bandwidth. The SpMV multiplication is a prime example of such a bandwidth-bound computation.

While vectorisation increases the maximum computational throughput of CPUs, it does nothing to alleviate bandwidth problems. However, early observations regarding SpMV multiplication on the Xeon Phi, such as by Saule et al. [1], show that the SpMV multiplication is *latency* bound instead of bandwidth bound. This was also observed for one of the best performing methods for SpMV multiplication by Yzelman and Roose [2], the row-wise 1D distributed coarse-grained method. If the cause lies with threads not issuing enough memory requests to saturate the memory architecture, vectorisation actually can help to increase performance, even for applications that are commonly considered bandwidth bound. For the SpMV multiplication, the efficient use of vectorisation is made possible by a pair of vector instructions included with the Intel Xeon Phi instruction set: the gather and the scatter instructions.

The efficiency of SpMV multiplication depends strongly on the nonzero structure and dimensions of the input matrix, but also strongly depends on the target architecture specifics. This work investigates the performance of the SpMV multiplication by benchmarking a wide set of matrices across three fundamentally different architectures: the Intel Xeon Phi, a dual-socket Ivy Bridge machine, and the Nvidia Tesla K20X GPU.

The remainder of this section introduces standard methods and notation for SpMV multiplication and present an efficient parallelisation scheme. Section II explains the preferred data structure for sparse matrix multiplication in more detail, and Section III proceeds on how to incorporate vectorisation into that scheme. The resulting vectorised BICRS data structure is compared to earlier work in Section IV, and discusses in which way the scheme generalises existing data structures. Section V tests the method in practice and compares it to other state-of-the-art methods and architectures, and clearly demonstrates the performance tradeoffs due to vectorisation on the Intel Xeon Phi. Section VI concludes and suggests future work.

A. Standard methods for SpMV multiplication

Given an $m \times n$ input sparse matrix A that contains nz nonzeroes, an $n \times 1$ input vector x, and an $m \times 1$ output vector y. The matrix–vector multiplication $y = Ax$ computes $y_i = \sum_{j=0}^{n-1} a_{ij} x_j$ for all $0 \leq i < m$. While dense multiplication has a computational complexity of $\Theta(mn)$, sparse matrix–vector multiplication is instead characterised by a running time of $\mathcal{O}(nz)$. The coordinate (COO) data structure stores each nonzero separately, thus attaining the $(2 \cdot b_{\text{index}} + b_{\text{value}})nz$ memory bound, with b_{index} and b_{value} the number of bytes required to store a single (row or column) coordinate and a single nonzero value, respectively .

Let the nonzeroes in A be numbered arbitrarily, so that the kth nonzero value $\boldsymbol{v}_k = a_{\boldsymbol{i}_k \boldsymbol{j}_k} \in A$ can be stored in the kth entry of three arrays $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{v}$, $0 \leq k < nz$. A single COO SpMV multiplication then proceeds in $\Theta(nz)$ time as follows:

for $k = 0$ **to** $nz - 1$ **do**
 $y_{\boldsymbol{i}_k} := y_{\boldsymbol{i}_k} + \boldsymbol{v}_k \cdot x_{\boldsymbol{j}_k}$

The Compressed Row Storage (CRS) data structure[1] [5], the defacto standard for sparse computations, requires $(nz + m + 1)b_{\text{index}} + b_{\text{value}} nz$ storage capacity and attains a run time of $\Theta(nz + m)$. In parallel shared-memory computation, the performance increase of CRS over COO is much higher than the reduced run time complexity

[1] CRS was earlier known as the Yale format [3], and currently is also known as the Compressed Sparse Rows (CSR) [4].

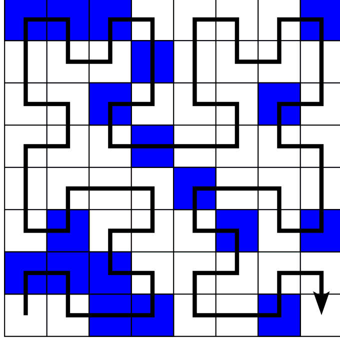

Figure 1. Regular sparse blocking in Hilbert order. Each coloured square corresponds to a submatrix of size $\beta \times \beta$, with $\beta \ll m, n$.

would indicate. This is due to saving $b_{\text{index}}(nz - m)$ bytes in storage; any reduction of the required data movement immediately translates to an increase of the bandwidth-bound SpMV multiplication kernel.

B. Advanced methods for SpMV multiplication

The sparse matrix A can be blocked into submatrices of size $\beta \times \beta$. Unlike with dense computations, blocking alone will not increase cache efficiency [6]. Using the Hilbert curve to order the blocks, however, does increase efficiency through cache-oblivious means [7]; this is illustrated in Figure 1. The nonzeroes within each block have a row-major order, which helps reduce the memory storage requirement [2].

Using CRS to store the nonzeroes in each $\beta \times \beta$ block will require a total of $\Theta(nz + nm/\beta)$ memory and thus does not scale. COO is a better choice as the total storage requirements remains at $2b_{\text{index}}nz + b_{\text{value}}nz$ and does not increase when storing a matrix block-by-block.

The data in the arrays $\boldsymbol{i}, \boldsymbol{j}$ may be further compressed to use $2\lceil \log_2 \beta \rceil nz$ bytes instead of $2b_{\text{index}}nz$. Applying this compression requires an auxiliary data structure to record the row- and column-wise starting position of each of the $\beta \times \beta$ submatrices of A. The Compressed Sparse Blocks (CSB) scheme uses CRS as this auxiliary data structure, and proves that under reasonable conditions the total data structure storage requirements equal that of non-blocked CRS of A [8].

The non-scalability of CRS has earlier been observed in distributed-memory SpMV multiplication, and eventually led to the Bi-directional Incremental Compressed Row Storage (BICRS) data structure [9]. BICRS has a storage requirement of $(nz + \mathit{jumps})b_{\text{index}} + b_{\text{value}}nz)$, with *jumps* the number of row-wise jumps in the imposed ordering of the nonzeroes of A; this value is tightly bounded by nz, and if nonzeroes are in a row-major order, this value is tightly bounded by m; BICRS thus is a strict improvement on both COO and CRS, while multiplication still proceeds in $\Theta(nz)$ time.

Yzelman and Roose [2] combine the insight of CSB with BICRS to improve on both, and obtain a data structure that has an (unlikely) worst case to equal that of blocked CSB and non-blocked CRS. It arises in a similar way as CSB is constructed; the sparse matrix is blocked, BICRS is used instead of COO to store nonzeroes within each block, resulting in a $\Theta(\log_2 \beta)$ memory requirement per BICRS array element. BICRS also substitutes CRS as the auxiliary data structure to store the blocks themselves, thus enabling arbitrary block traversals necessary for Hilbert ordering. In the worst case, the total memory requirement of this compressed BICRS (CBICRS) scheme equals $(nz + m)b_{\text{index}} + b_{\text{value}}nz$, while usually it is lower. The time complexity of SpMV multiplication remains $\Theta(nz)$.

C. Parallelisation

The parallelisation scheme of choice is a 1D row-wise distributed scheme executed in a coarse-grained fashion to maximise data locality. The advantage of performing a 1D parallel SpMV multiply is that no barrier synchronisations are required; this makes a huge difference on architectures like the Intel Xeon Phi, where synchronisation is costly.

The rows of A are split in P contiguous parts, with splitting points $s_1 \leq s_2 \leq \ldots \leq s_{p-1}$, and P the number of available hardware threads. The starting point s_0 and ending point s_P are set to 0 and m, respectively. A greedy load-balancing scheme chooses the smallest possible integer s_{k+1} for which $\sum_{i=s_k}^{s_{k+1}} nnz(A_i) > \frac{nz}{P}$, starting with $k = 1$ up to $k = P-1$. Thread k processes all nonzeroes on the rows s_k to $s_{k+1} - 1$, and stores these nonzeroes locally; the output vector elements y_{s_k} through $y_{s_{k+1}-1}$ can thus be kept in local memory and benefit from fast local access, while the input vector must instead be accessed globally and thus is stored in a single contiguous memory chunk in shared memory.

In summary, this scheme first pre-processes the input matrix A into thread-local matrices A_i and thread-local output vectors $y^{(i)}$. A parallel SpMV multiplication then simply consists of P threads locally executing the sequential SpMV multiplication $y^{(i)} = A_i x$, where each thread-local A_i contains less rows than m but retain the full number of columns n. Other approaches are feasible too, for instance a fine-grained scheme or a fully distributed scheme, although the first would lose data locality while the latter suffers from high synchronisation costs on this architecture.

For a recent overview of general high-level approaches to shared-memory parallel SpMV multiplication, see Yzelman and Roose [2]. This paper will assume this row-wise 1D scheme to demonstrate the performance of the vectorised data structure; when using a different parallelisation strategy, the vectorised data structure presented here can still be applied.

II. Compressed BICRS

Considering the high-level requirements of an efficient shared-memory parallelisation of sparse computations [2], any efficient data structure for SpMV multiplication requires support for

- arbitrary-order nonzero traversal,
- sparse blocking, and
- index compression.

The CBICRS data structure achieves precisely this, and top-down optimisation thus suggests to modify this data structure to allow for vectorised SpMV multiplication. To be able to do so, this section shall first describe CBICRS in full detail.

Define the *delta encoding* of the array $\boldsymbol{i}$ as:

$$\mathit{delta}(\boldsymbol{i})_k = \begin{cases} \boldsymbol{i}_0 & \text{for } k = 0, \\ \boldsymbol{i}_k - \boldsymbol{i}_{k-1} & \text{for } 0 < k \leq nz; \end{cases}$$

that is, $\mathit{delta}(\boldsymbol{i})$ is an array in which the first element equals that of $\boldsymbol{i}$, while subsequent element encode the difference between current and subsequent positions. The delta encoding $\mathit{delta}(\boldsymbol{j})$ is defined similarly. Note that for this delta encoding, successive nonzeroes will appear whenever a string of nonzeroes share the same row or column. In such cases, zero-length encoding will further reduce memory requirements:

$$\mathit{zle}(\mathit{delta}(\boldsymbol{i}), \mathit{delta}(\boldsymbol{j})) = (\boldsymbol{i}^{\text{bicrs}}, \boldsymbol{j}^{\text{bicrs}}), \text{ with}$$

$$\boldsymbol{i}^{\text{bicrs}} = (delta(\boldsymbol{i})_0, delta(\boldsymbol{i})_{1:nz}\backslash\{0\})$$

$$\boldsymbol{j}_k^{\text{bicrs}} = \begin{cases} delta(\boldsymbol{j})_k + n & \text{if } delta(\boldsymbol{i}_k) \neq 0 \\ delta(\boldsymbol{j})_k & \text{otherwise.} \end{cases}$$

For example, considering the first nine nonzeroes of Figure 1 while assuming for brevity that $\beta = 1$ and $m = n = 8$:

$$\begin{aligned} zle((6,0,1,0,\text{-}1,\text{-}1,\text{-}5,0,0) \quad &, \quad (0,1,1,1,\text{-}1,\text{-}1,\text{-}1,1,1)) \\ = ((6,1,\text{-}1,\text{-}1,\text{-}5) \quad &, \quad (0,1,9,1,7,7,7,1,1)). \end{aligned}$$

SpMV multiplication with BICRS uses pointers to x and y for efficiency, for which the code is given in Algorithm 1. To demonstrate the elegance of using pointer arithmetic in this code, short C code annotations as used in the implementation are included as well.

When A is blocked into submatrices of size $\beta \times \beta$, then elements of $\{\boldsymbol{i}, \boldsymbol{j}\}^{\text{bicrs}}$ need at most $3\lceil \log_2 \beta \rceil$ bytes each, or $2\lceil \log_2 \beta \rceil$ if nonzeroes in each block are in row-major order. Using smaller raw data types such as `char` or `short int` to achieve this lower storage requirement in implementation leads to the Compressed BICRS data structure. In implementation, $\beta = 16384$ which enables using 16-byte integers to store the arrays $\{\boldsymbol{i}, \boldsymbol{j}\}^{\text{bicrs}}$.

The SpMV multiplication kernel *per submatrix* is given by Algorithm 1. How to achieve the per-block Hilbert traversal is described and analysed in detail in the article by Yzelman and Roose [2] and its supplement[2].

Algorithm 1 SpMV multiplication using BICRS

1: Let x, y, $\boldsymbol{i}^{\text{bicrs}}$, $\boldsymbol{j}^{\text{bicrs}}$, and $\boldsymbol{v}$ be pointers to their arrays
2: Set x_e to x shifted by n {`x_e=x+n;`}
3: Set v_e to v shifted by nnz {`v_e=v+nnz;`}
4: Shift x by $\boldsymbol{j}^{\text{bicrs}}$, and increment the latter {`x+=*j++;`}
5: **while** $v < v_e$ **do**
6: Shift y by $\boldsymbol{i}_j^{\text{bicrs}}$ and increment $\boldsymbol{i}_j^{\text{bicrs}}$ {`y+=*i++;`}
7: **while** $x < x_e$ **do**
8: $y := y + v \cdot x$ and increment v {`*y+=*v++**x;`}
9: Shift x by $\boldsymbol{j}^{\text{bicrs}}$ and increment $\boldsymbol{j}^{\text{bicrs}}$ {`x+=*j++;`}
10: Shift x by $-n$ {`x-=n;`}

III. Incorporating vectorisation

Many modern architectures have hardware support for vectorisation; this enables using one operation (e.g., addition) to operate on two vectors instead of on two scalars only, thus increasing the peak computational performance: if a vector contains 4 (64-bit) floating point numbers, the maximum floating point operations per second (flop/s) effectively quadruples. The vector size of machines supporting the advanced vector extensions (AVX) indeed already is 256 bit, while AVX-512 will double this size to 512 bits.

To use vector instructions, l elements must be loaded from main memory into vector registers first. Loading l contiguous values into such a register from a cache-aligned memory location results in high-throughput data streams for which a latency cost ideally appears only once per stream. With sparse computations, however, indirect accessing is the norm. The *gather* and *scatter* operations, included with the Intel Xeon Phi instruction set, provides a solution: given an in-memory array $\boldsymbol{v}$ and an integer vector register $r_j = (i_0, \ldots, i_{l-1})$, 'Gather($r_i$,$\boldsymbol{v}$,$r_j$)' loads $(\boldsymbol{v}_{i_0}, \ldots, \boldsymbol{v}_{i_{l-1}})$ into vector register r_i. There is no guarantee on the maximum latency of a gather instruction. 'Scatter($\boldsymbol{v}$,r_i,r_j)' does the inverse and writes $(r_i)_k$ to $\boldsymbol{v}_{(r_j)_k}$ for all $0 \leq k < l$. The gather will also be available in the AVX2 instruction set, while the scatter will be available from AVX-512 on. The vectorisation length l is related to the data type and the vector register size; this paper assumes 64-bit floating point numbers, hence $l = 8$ on the Intel Xeon Phi.

To allow for vectorisation within the CBICRS data structure, the inner kernel of the SpMV multiplication, $y_i := a_{ij}x_j$, must be modified to operate on vectors of length l instead. This means l elements from $\boldsymbol{v}$ and $\boldsymbol{x}$ must be read into vector registers r_2 and r_3, respectively, so that the innermost SpMV multiplication kernel can be rewritten to

$$r_1 := r_1 + r_2 \odot r_3, \tag{1}$$

with r_1 an additional output vector register, and '$\odot$' (the Hadamard product) an element-wise multiplication. Many modern architectures support such a *fused multiply-add* (FMA, Eq. 1) in hardware, resulting in one (pipelined) vectorised FMA per CPU clock cycle, i.e., $2l$ floating point operations (flops) per cycle.

The register r_2 will contain l unique nonzero values, but elements of r_1 do not have to correspond to a contiguous range of elements from y, nor even to unique elements from y. If all elements in r_1 correspond to a single y_i, then by nature of the SpMV multiplication the elements of r_3 will have to correspond to l different elements from x. Likewise, if r_1 corresponds to 2 unique elements from y, then r_3 will have to correspond to at least $l/2$ different elements from x, and so on: if p is the number of unique elements from y corresponding to elements stored in r_1, then $q = l/p$ is the minimum number of unique elements from x that correspond to elements in r_3. This observation leads to the notion of $p \times q$ *blocking for vectorisation*.

A. Vectorised BICRS

Let $(\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{v})$ correspond to the COO data structure of the nonzeroes of A ordered such that their $bl+k$th entries (i.e., $\boldsymbol{i}_{bl+k}$ and so on), for a given integer $0 \leq b < nblks$ and all integers $0 \leq k < l$, contains the l nonzeroes assigned to the bth vectorisation block of A. The ordering of nonzeroes must not break the condition $p \times q$ blocking for vectorisation as described above.

Let the arrays $\boldsymbol{i}_{\text{vblk}}$ and $\boldsymbol{j}_{\text{vblk}}$ of length $nblks$ store the coordinates of the first nonzero of each of the $nblks$ vectorisation blocks, i.e., those nonzeroes with index bl, for all $0 \leq b < nblks$. Let $\boldsymbol{i}_{\text{vblk}}^{\text{bicrs}}$ and $\boldsymbol{j}_{\text{vblk}}^{\text{bicrs}}$ be the BICRS encoding of those two arrays. To have the gather and scatter operations work together with BICRS pointer arithmetic on *all* nonzeroes of A, the positions of remaining nonzeroes in each vectorisation block are stored relative to the coordinate of the first nonzero in each block. Let $\boldsymbol{i}^{\text{vecbicrs}}$ and $\boldsymbol{j}^{\text{vecbicrs}}$ be the index arrays of the modified BICRS data structure. Then

$$\boldsymbol{i}_{bl+k}^{\text{vecbicrs}} = \begin{cases} (\boldsymbol{i}_{\text{vblk}}^{\text{bicrs}})_b & \text{if } k = 0 \\ \boldsymbol{i}_{bl+k} - \boldsymbol{i}_{bl} & \text{if } 1 \leq k < l, \end{cases} \tag{2}$$

$$\boldsymbol{j}_{bl+k}^{\text{vecbicrs}} = \begin{cases} (\boldsymbol{j}_{\text{vblk}}^{\text{bicrs}})_b & \text{if } k = 0 \\ \boldsymbol{j}_{bl+k} - \boldsymbol{j}_{bl} & \text{if } 1 \leq k < l, \end{cases} \tag{3}$$

for all $0 \leq b < nblks$. Note that these relative indices allow for the same compression techniques as used with regular BICRS, and allow for an arbitrary order of the $p \times q$ blocks, and allow for an arbitrary order of nonzeroes *within* each block; all as long as the constraint of having a minimum of q different column indices when handling a vector block with p unique rows holds true.

[2]The supplement is freely available from IEEE, via http://ieeexplore.ieee.org/ielx7/71/6674937/6463397/ttd2014010116.zip.

B. Multiplication using vectorised BICRS

The BICRS SpMV multiplication algorithm from 1 is adapted to incorporate vectorisation, resulting in Algorithm 2. First, the inner kernel is vectorised as per Eq. 1, as shown on line 9. This requires the vector registers $r_{\{1,2,3\}}$ be filled with possibly overlapping output vector elements, unique nonzero values, and possibly overlapping input vector elements, respectively.

Algorithm 2 Pseudocode for vectorised SpMV multiplication.

```
1:  Let x, y, i^vecbicrs, j^vecbicrs, and v be pointers to their arrays
2:  Load (r6, i^vecbicrs), load (r4, j^vecbicrs), set x_e = x + n, set b_i = 0
3:  Shift x with (r4)_0, shift y with (r6)_0
4:  Gather (r5, y, r6), set (r4)_0 = 0, set (r6)_0 = 0
5:  while there are blocks left do
6:      Set r1 to all zeroes
7:      while x < x_e do
8:          Gather (r3, x, r4), load (r2, v)
9:          Set r1 := r1 + r2 ⊙ r3 (vectorised FMA)
10:         Load (r4, j^vecbicrs)
11:         Shift x with (r4)_0
12:         Set (r4)_0 = 0
13:     Add sum_{j=0}^{p-1} (r1)_{b_i p + b_j} to (r5)_{b_j + b_i q}, ∀ 0 ≤ b_j < q (reduce)
14:     Shift x with −n, increment b_i
15:     if b_i equals p then
16:         Scatter (y, r5, r6)
17:         Load (r6, i^vecbicrs)
18:         Shift y with (r6)_0
19:         Set (r6)_0 = 0
20:         Gather (r5, y, r6), set b_i = 0
```

Elementary vector operations on vector registers $r_0, \ldots, r_6$ of size l:
'Load (r_i, x)': load the next l elements from x into r_i (streaming load).
'Gather (r_i, x, r_j)': gathers l elements from x with offsets r_j into r_i.
'Scatter (y, r_i, r_j)': scatters l elements from r_i into y with offsets r_j.

The data required for r_2 can be streamed directly from the vectorised BICRS data structure, as on line 8. The elements of $r_{\{1,3\}}$ must be derived from $\{\boldsymbol{i}, \boldsymbol{j}\}^{\text{vecbicrs}}$, however, which first requires reading in those two arrays on two auxiliary vector registers $r_{\{4,5\}}$ as on lines 2, 10, and 17. Since now the first elements of $r_{\{4,5\}}$ correspond to the regular BICRS format, the pointer shifts to the input and output vector should be handled accordingly. Afterwards, the first entries of $r_{\{4,5\}}$ are set to zero to enable a subsequent gather. These two-line operations are shown starting on lines 3, 11, 18. Row jumps are handled as with BICRS via the lines 7 and 14.

Setting the first elements of $r_{\{4,5\}}$ to zero enables the use of the gather instruction to retrieve $r_{\{1,3\}}$ using $r_{\{4,5\}}$; see lines 4, 8 and 20. Vectorised BICRS does not issue gathers (and scatters) on output vector elements at every $p \times q$ vectorisation block if the current set of a maximum of p unique rows are also used by the next vectorised block. Instead, when a new set of rows are gathered they are put in another auxiliary array r_6, while r_1 is set to a zero-vector. Once a change in the set of rows is detected, r_1 is reduced into r_6 on line 13, and r_6 is scattered back into main memory on line 16.

The reduction of r_1 into r_6 is necessary because the number of unique row elements corresponding to its elements may be less than l; and by construction of the vectorised BICRS data structure, the number of unique row elements is exactly p. If the map π from elements in r_1 to elements in y indeed is not injective (i.e., if $p < l$), then the elements in r_1 that map to the same element of y must first be reduced before a scatter operation can proceed, or data races will occur.

This (partial) reduction $(r_6)_k = \sum_{i \in \pi^{-1}(\pi(k))} (r_1)_i$ is handled on line 13, which, in implementation, performs $\log_2 q$ steps of vectorised permutations and additions. On the Intel Xeon Phi, efficient implementation of this strategy also required a per-block permutation of the $\{\boldsymbol{i}, \boldsymbol{j}\}^{\text{vecbicrs}}$ vectors as there is no hardware support to permute vector register elements past the middle 256-bit boundary. As discussed, the scatter of r_6 is delayed until the register stores precisely l unique and correctly reduced elements that will not be used to process a subsequent vectorisation block.

If $p = 1$ then all elements in r_1 correspond to the same vector element, thus making gather/scatter instructions on y unnecessary; instead, when a row jump is detected, the allreduce $\alpha = \sum_{i=0}^{l-1} (r_1)_i$ is executed and α is directly added to the correct element of y.

If $p = l$ then all elements in r_1 correspond to different output vector elements. The gather on y can then proceed directly on r_1, while on row changes the scatter can directly operate using r_1 without the need of an intermediate (partial) reduction. Hence choosing $p = l$ is preferable computationally, but recall that computation is not the bottleneck in SpMV multiplication: choosing different p may lead to more efficient use of the memory hierarchy, and thus may be preferable over $p = l$ regardless of its increased computational cost.

C. Constructing suitable nonzero traversals

For each vectorisation block the row and column indices are fixed, but need not be contiguous. This allows for arbitrary traversals to a wide extent: successive nonzeroes can be added in-turn to the blocks until (1) a nonzero is to be added that has a row index that does not appear in the block while the block already has the full number of p distinct rows, or (2) the nonzero corresponds to a row that already has q nonzeroes in the current block. In case of such an overflow, the block will be completed by filling any remaining capacity with explicit zeroes, thus resulting in *fill-in*. The overflowing nonzero will then be added as the first nonzero in a next $p \times q$ block.

The above sketches a greedy construction which suits the higher-level scheme used in the experiments: the matrix A is blocked in $\beta \times \beta$ submatrices that are ordered according to the Hilbert curve, while nonzeroes within each submatrix are ordered in a row-major fashion. Following the above procedure, p rows of each subblock are handled one after the other. Write nz_i as the number of nonzeroes contained on the ith row within such a row group, then there shall be $k = (\max_{i \in \{0,1,\ldots,p-1\}} nz_i)/q$ vectorisation blocks corresponding to this row group. If the number of nonzeroes differ within rows in a single row group, then the fill-in is nonzero and equal to $kq - \sum_{i=0}^{p-1} nz_i$.

A good nonzero traversal for vectorisation is one that minimises fill-in; minimised fill-in results in minimised memory footprints and thus results in minimal taxing of the memory hierarchy. Assuming the greedy ordering method described here, then there remains a single parameter that may be used to tune fill-in, namely, p.

IV. In relation to earlier work

The vectorised BICRS data structure has close ties to three well-known families of approaches for the vectorisation of sparse matrix computations: (1) segmented scans, (2) ELLPACK, and (3) Blocked CRS. All three formats were well-known but have been attracting renewed attention due to their applicability to general purpose GPU computing.

A. Segmented scan

Blelloch et al. first used segmented operations for sparse computations on vector computers [10]. A segmented SpMV multiplication

proceeds in three stages: (1) for all nonzeroes, multiply $a_{ij}x_j$; (2) reduce the resulting nz-length vector to an m-length vector by summing elements on the same row i, and (3) add the reduced values to the appropriate elements of y.

Step 2 is a segmented reduce, and requires that the vector is sorted by row, i.e., the nonzeroes should appear in row-major order. Segmented operations use bitmask arrays to indicate the row boundaries of the reduction. In the vectorised BICRS regular $p \times q$ blocking with fill-in is used instead, though other than that, taking $p = 1$ follows the same procedure as that of Blelloch et al.

B. ELLPACK

As with multiplication based on segmented operations, ELLPACK was likewise designed for sparse computations on vector computers [11][3]. If A has k elements on each row, then ELLPACK performs an SpMV in k steps, where each step performs (1) retrieval of the tth nonzero a_{ij} on each row i, and (2) addition of $a_{ij}x_j$ to y_i. This approach is parallel in the row direction, but the restriction of having a constant number of elements t on each row means fill-in will be abundant on general sparse matrices.

The *sliced* ELLPACK (SELLPACK) data structure limits the range of the rows to a pre-determined slice length s; A then only a requires a constant number of nonzeroes per row, every s rows. Sorting strategies, such as simply ordering the rows of A according to the number of nonzeroes each row contains, will naturally lead to limited fill-in. Note that such reordering techniques will benefit vectorised BICRS when $p > 1$, and note that for $p = l$, vectorised BICRS exactly coincides with SELLPACK with $s = l$.

C. Blocked CRS

Blocked CRS (BCRS) modifies CRS to store fixed-size blocks ($p \times q$) instead of individual nonzeroes [12]. The motivation for BCRS was *not* vectorisation, but rather additional compression; when blocks instead of nonzeroes are stored, the number of index array elements per nonzero is reduced beyond that of the CRS data structure. Like with BICRS, the reduced bandwidth pressure directly translates to an increased SpMV multiplication speed.

The $p \times q$ blocks in BCRS correspond to rectangular blocks in A; the p rows in a single block must be contiguous in A, as must be the q columns assigned to a single block. In contrast, the $p \times q$ blocks in vectorised BICRS do not correspond to rectangular blocks in A; the nonzeroes in a single block must reside on any row as long as there are p distinct rows, while the column indices are completely freely chosen.

D. Other formats

All three formats have seen extensions since their introductions. Segmented scans in sparse computations have appeared in shared-memory parallel contexts [13] and on GPU-tailored structures [14], [15]. Segmented-scan like operations also are required to implement COO-based SpMV multiplication on GPUs, which may simultaneously be augmented with the use of ELLPACK, as, e.g., done in the popular hybrid COO/ELLPACK format (HYB) [16]. Applying the same data structure several times for different parts of the matrix may also be helpful and led to SELLPACK and auto-tuners such as OSKI [17]. Modern approaches to sorting for SELLPACK have appeared and attempt to move beyond GPU computing [18]. SpMV multiplication based on segmented operations or on SELLPACK can both be implemented using standard CRS [10], [19].

[3]There may be earlier references than Rice & Boisvert (1985) describing ELLPACK, but none seem accessible at present.

Virtually all formats can additionally benefit from sparse matrix reordering, for various reasons: such as creation of small dense blocks for BCRS to exploit [20], fill-in reduction in SELLPACK, or provable minimisation of cache-misses during SpMV multiplication [7], [21]. Reordering has many other advantages in sparse computations as well [22], [23], [24].

None of the existing methods described a vectorisation scheme that mixes segmented reductions ($p = 1$), ELLPACK ($p = l$), and BCRS (rectangular blocks). Additionally, none of the existing schemes allows for vectorisation simultaneously with advanced compression, sparse blocking, and arbitrary nonzero traversals. The vectorised BICRS lays bare the new dimension $0 < p < l$ in which sparse computations may be vectorised, orthogonal to the many other possible optimisation techniques that are appearing in literature.

V. Experimental evaluation

The vectorised compressed BICRS data structure and its corresponding SpMV multiplication kernel have been implemented on the Intel Xeon Phi model 7120A, using the various intrinsics delivered with the Intel C++ Compiler (ICC) 14.0.1. The code is written in ANSI C++ and is freely available[4]. The parallelisation scheme is as described in Section I-C, with thread-local multiplication augmented using sparse blocking and Hilbert ordering as described in Section I-B and by Yzelman and Roose [2], implemented using POSIX Threads.

The performance is compared to the baseline implementation that uses regular non-vectorised compressed BICRS on both the Intel Xeon Phi as well as a dual-socket Intel Xeon 'Ivy Bridge' machine. An additional baseline comparison is provided by the hybrid ELLPACK/COO format [16] which is provided with the CuSparse CUDA package, as run on an NVIDIA Tesla K20X. The input matrices tested on come from different classes of input matrices that come from a wide range of applications.

A. Dataset selection

Previous studies have shown sparse matrices that already possess a cache-friendly layout of nonzeroes behave very differently from matrices that have no beneficial structure [7], [9]. Therefore, an initial classification of datasets uses the notion of structured versus unstructured matrices; an example of the former is a matrix consisting solely of several dense diagonals (such as s3dkt3m2), while an example of the latter may arise from the investigation of parasitic influences during chip design (memplus) or from investigating hyperlink structures (the wikipedia matrices).

A second division in category is made on basis of the dimensions of the sparse matrix: if m, n are small, then the corresponding input and output vectors can be cached completely, thus resulting in much faster memory accesses and high performance during SpMV multiplication. The size in terms of nonzeroes is of much less import; nonzero elements are read only once, so caching them will not increase performance.

Table I shows the matrices selected for this benchmarking effort. All matrices are freely available from the University of Florida Sparse Matrix Collection [25][5]. A balance is struck between structured and unstructured matrices, and each category contains both small matrices (in-cache vectors) as well as large matrices (out-of-cache vectors). Other considerations were to choose matrices also used in earlier studies [1], [2], [7], [8], [9].

[4]See http://albert-jan.yzelman.net/software#SL.

[5]except for the tbdlinux matrix which can be obtained from http://www.math.uu.nl/people/bisseling/Mondriaan/.

#	*Structured matrices*	Size ($m+n$)	Nonzeroes (nz)
01	fidap037	7 130	67 591
02	s3rmt3m3	10 714	207 123
03	bcsstk17	21 948	428 650
04	memplus	35 516	99 147
05	lhr34	70 304	746 972
06	bcsstk32	89 218	1 029 655
07	s3dkt3m2	180 898	1 921 955
08	RM07R	763 378	37 464 962
09	Emilia_923	1 846 272	40 373 538
10	cage14	3 011 570	27 130 349
11	cage15	10 309 718	99 199 551
12	adaptive	13 631 488	13 624 320
#	*Unstructured matrices*		
13	lp_qap15[†]	28 605	94 950
14	rhpentium_new	50 374	258 265
15	andrews	120 000	760 154
16	tbdlinux[†]	132 924	2 157 675
17	nd24k	144 000	28 715 634
18	nug30[†]	431 610	1 567 800
19	stanford_berkeley	1 366 892	7 583 376
20	ldoor	1 904 406	23 737 339
21	wikipedia-2005	3 269 978	19 753 078
22	wikipedia-2006	6 296 880	39 383 235
23	wikipedia-2007	7 133 814	45 030 389
24	hugetric-00020	14 245 584	21 361 554

Table I
MATRICES USED IN EXPERIMENTS, SORTED ACCORDING TO DIMENSION SIZE. DAGGERS INDICATE NON-SQUARE MATRICES.

#	1×1	1×8	2×4	4×2	8×1	BW	
01	12.1	**19.9**	19.9	14.6	9.86	171	312
02	13.1	29.1	36.0	**36.5**	31.5	201	467
03	14.2	**30.7**	30.3	28.8	24.9	169	390
04	8.4	7.8	**12.7**	9.5	5.3	120	203
05	12.7	19.3	20.0	**21.1**	18.4	196	347
06	14.1	28.0	**31.6**	30.8	26.2	165	396
07	13.4	26.6	30.0	**31.9**	30.4	156	389
08	12.1	30.1	**32.1**	24.1	15.7	136	323
09	11.2	19.2	20.8	**20.8**	20.0	114	267
10	7.4	5.7	8.4	10.8	**12.3**	86	156
11	7.0	-	7.7	9.8	**11.1**	75	140
12	4.4	2.9	3.9	4.5	**4.7**	67	84
#	1×1	1×8	2×4	4×2	8×1	BW	
13	10.9	16.6	17.4	**18.4**	15.6	128	245
14	1.2	**2.4**	1.4	0.7	0.3	28	45
15	**5.8**	3.56	4.3	4.5	4.3	43	72
16	8.0	**10.9**	8.9	5.8	3.3	85	155
17	12.3	**28.5**	28.4	27.2	25.1	141	353
18	6.5	3.2	4.6	6.1	**6.7**	49	91
19	4.8	**7.5**	5.8	3.8	2.1	58	110
20	11.0	19.4	**19.6**	19.3	18.0	108	251
21	**2.5**	1.6	1.9	2.0	2.1	37	48
22	**1.9**	1.2	1.5	1.5	1.6	32	42
23	**1.7**	1.2	1.4	1.4	1.5	31	40
24	**1.3**	0.9	1.1	1.1	1.2	29	35

Table II
SPEED IN GFLOP/S OF SPMV MULTIPLICATION USING VECTORISED CBICRS ON AN INTEL XEON PHI. THE 1×1 BLOCK SIZE CORRESPONDS TO NON-VECTORISED CBICRS. THE BEST SPEEDS ARE PRINTED IN BOLD FACE. THE RIGHT-MOST COLUMN SHOWS THE LOWER BOUND (LEFT) AND UPPER BOUND (RIGHT) ON EFFECTIVE BANDWIDTH IN GBYTE/S.

B. *Vectorisation results*

The Xeon Phi used has $P=240$, $l=8$, 16 GB of memory with 352 GB/s peak bandwidth, and 512 kB L2 caches shared among four hardware threads per each of the 60 cores. Table II shows the attained effective performance (Gflop/s) per matrix and per block size, where the 1x1 category corresponds to using non-vectorised CBICRS. The speeds are obtained by performing 10 benchmarks, where a single benchmark consists of one SpMV multiplication to warm up the caches, followed by a timed run of at least 10 000 multiplications. From the computation speed, an upper bound on the attained effective bandwidth can be calculated by assuming all vector elements, once cached, remain cached; likewise, a lower bound may be deduced by assuming no caching at all. Both bounds are included in the table.

For the structured matrices, vectorisation convincingly helps to reduce the latency of indirect vector addressing for all matrices, in the best case attaining a 2.78x speedup over non-vectorised code (the s3rmt3m3 matrix using 4×2 blocking). The highest gain on unstructured matrices is 2.32x (nd24k, 1×8 blocking). Vectorisation is visibly less effective for unstructured matrices, where only half of those matrices benefited from vectorisation, with virtually none of the larger matrices being sped up.

The latter is not surprising when looking at the fill-in incurred for matrices 21–23: vectorisation there increases the memory footprint by at least 112 percent, and reaches up to a 448 percent increase for wikipedia-2007 using 1×8 blocking. The worst fill-in was incurred for again $p=1$ on the cage15 matrix, which caused the Xeon Phi to run out of memory. The largest matrix, hugetric-00020, attained a maximum fill-in of 26 percent increase only, also for $p=1$. The lowest measured increased is 1.2 percent, attained on the s3dkt3m2 for $p=4$; as one might expect, this also leads to the best performing blocking scheme for that matrix in terms of speed.

The heuristic that a minimised fill-in leads to best performance holds in 10 of the 20 cases in which vectorisation was effective. The maximum relative performance loss due to using the heuristics anyway is 43 percent (rhpentium, choosing $p=4$ instead of $p=1$), while the maximum absolute performance loss is 3.2 Gflop/s (memplus, choosing $p=4$ instead of $p=2$). Due to space considerations, the exact fill-in measures are omitted from this paper, but may freely be requested from the author.

C. *In comparison to other architectures*

The shared-memory machine compared against contains two Intel Xeon E5-2690 v2 (Ivy Bridge) processors, each consisting of 10 cores each with a private 256 kB L2 cache, and all cores sharing a 25 MB L3 cache [26]. Each socket connects four 16 GB DDR3 modules operating at a total memory capacity of 128 GB and a theoretical maximum bandwidth of 51.2 GB/s. Note that Ivy Bridge does not have support for the gather and scatter instructions needed for the vectorisation approach presented here. The software, thus using non-

vectorised CBICRS, was compiled and optimised by GCC 4.6.4[6].

The NVIDIA Tesla K20X GPU has 14 streaming multi-processors (SMXs) that each contain 192 CUDA cores, for a total 2688 cores. Each SMX processor has a 64 kB L1 cache available, and all processors share a 1.5 MB L2 cache. It has 6 GB on-board GDDR5 memory with a maximum bandwidth of 250 GB/s. The official HYB implementation of the cuSPARSE library that comes with CUDA 5.5 was used, and the driver program was compiled using GCC 4.8.2.

Table III compares the Ivy Bridge and K20X results to those of the Phi. The benchmarking methodology remains the same as described previously, with the additional remark that, like for the Xeon Phi, offload times are not included for the GPU timings as well. Both the Ivy Bridge and the Xeon Phi architectures perform very well on smaller dimension sizes, benefiting from the high-bandwidth and low-latency caches. While all architectures perform better on structured problems, the Xeon Phi has noticeable difficulty with unstructured matrices, incurring large performance penalties thanks to unpredictable memory accesses.

The benefit of the high K20X bandwidth capacity is clearly visible when processing larger matrices: multiplying using hugetric-00020 performs 5.2x faster on the K20X compared to the Ivy Bridge, which is indeed close to the relative difference between the theoretical maximum bandwidths of both architectures ($\frac{250\text{ GB/s}}{51.2\text{ GB/s}} = 4.9$). On most larger unstructured datasets, however, the relative performance difference is much less: the gain is in between 1–1.6x when considering matrices numbers 20–23.

D. In comparison to other vectorised methods

As suggested in Section IV, many of the known formats that are used for vectorisation are based on ELLPACK and/or segmented scans both of which this work generalises. A difference between the performance of vectorised BICRS compared to other work can be explained by either (1) an implementation error: the multiplication kernel in the other method can be rewritten to achieve comparable speed, or (2) a higher-level cause: schemes such as Hilbert-ordered blocking or row-wise reordering affect performance.

Thanks to the choice of overlapping datasets, a comparison of several approaches on the Xeon Phi can be given through literature. Saule et al. [1] implement a CRS-based approach that vectorises in the row direction ($p = 1$), and their experiments have three matrices in common: nd24k, ldoor, and cage14, with reported performances less than 23, 15, and 7 Gflop/s, respectively; the performance gain of vectorised BICRS over this method is 24 to 71 percent.

Liu et al. [27] provide a SELLPACK implementation ($p = l$) and have two datasets in common: ldoor and cage15. For ldoor they report 23 and 10 Gflop/s, respectively, when using various optimisation such as bitmasking, blocking, and/or reordering; for pure SELLPACK, they report 22.5 and 14 Gflop/s, respectively. The maximum gain on vectorised BICRS is 16 percent, and indicates that our implementation for $p = l$ can be improved and, at least for ldoor, can be augmented with the various techniques proposed by Liu et al[7].

Kreutzer et al. [18] have a ELLPACK-based format that includes row sorting, and included the RM07R matrix in their experiments. They attain 24 Gflop/s, for which vectorised BICRS performs 33 percent (for $p = 2$).

[6]The performance-related compilation flags were '-O3 -fprefetch-loop-arrays -funroll-loops -ffast-math -fno-exceptions'.

[7]the paper also introduces results using the K20X, but with an older version of CuSparse (5.0); using the version bundled with CUDA 5.5 significantly improves results: 25 and 35 percent for ldoor and cage15, respectively.

#	Xeon Phi		CPU	GPU
	best	*fill*		
01	20.0	19.9	**27.1**	2.5
02	**36.5**	36.0	29.9	6.5
03	30.7	**30.7**	33.0	8.6
04	12.7	9.5	**28.1**	3.7
05	21.1	19.3	**34.8**	10.3
06	31.6	31.6	**33.7**	21.3
07	31.9	31.9	**33.0**	27.5
08	25.8	**25.8**	17.7	25.6
09	20.8	20.8	16.6	**29.3**
10	12.3	12.3	13.0	**24.1**
11	11.1	11.1	10.9	**23.0**
12	4.7	4.7	3.9	**17.4**
#	Xeon Phi		CPU	GPU
13	18.4	16.6	**31.9**	12.5
14	2.4	1.4	**15.1**	5.9
15	5.8	4.5	**12.4**	10.4
16	11.0	11.0	**24.5**	12.6
17	28.5	28.5	18.5	**33.0**
18	6.7	6.7	**22.2**	13.4
19	7.5	7.5	**15.7**	12.4
20	19.6	19.4	15.9	**25.8**
21	2.5	2.0	7.4	**7.5**
22	1.9	1.5	4.7	**6.5**
23	1.7	1.4	4.4	**6.4**
24	1.3	1.2	2.6	**13.6**

Table III

SPEED OF SPMV MULTIPLICATION COMPARED ACROSS ARCHITECTURES. ALL NUMBERS ARE IN GFLOP/S. THE TWO FIGURES FOR THE XEON PHI REPORT THE BEST PERFORMANCE AS WELL AS THE PERFORMANCE WHEN CHOOSING p ACCORDING TO THE LEAST FILL-IN HEURISTIC.

VI. CONCLUSION

A new data structure for sparse matrices specifically designed for exploiting vectorisation capabilities was introduced: the vectorised (compressed) BICRS data structure. It uses the notion of $p \times q$ vectorisation blocks with $pq = l$ equal to the architecture vector length. The approach generalises various established formats, including those based on segmented scans ($p = 1$), Sliced ELLPACK ($p = l$), and Blocked CRS (non-rectangular blocks). It furthermore retains high compressibility, can perform sparse blocking, and can store nonzeroes in arbitrary orders. The concepts of irregular addressing, as is common in sparse computations, are merged with vectorised through the use of the gather and scatter operations.

Vectorised BICRS was tested for SpMV multiplication on the Intel Xeon Phi. Performance increased with 178 percent in one case and with 62 percent when averaging over all 24 matrices tested, when choosing the best performing p for each matrix. When instead choosing p according to the minimised fill-in heuristic, the average gain remains worthwhile at 54 percent. This clearly demonstrates that vectorisation can help improve performance applications with low arithmetic intensity such as the SpMV multiplication.

To compare the baseline performance, the vectorised multiplication as run on the Xeon Phi was compared to the performance of non-vectorised multiplication on an Ivy Bridge machine, and the vectorised ELLPACK/COO-based HYB scheme on the Nvidia K20X.

This comparison shows that the Xeon Phi performance is close to that of the Ivy Bridge for structured matrices, but falls behind on unstructured matrices, especially those of large dimensions. For larger matrices, the CPU and Xeon Phi benefit less from caching resulting in better performance for the GPU, especially on the larger structured matrices.

A. Future work

The presented strategy works with arbitrary nonzero orders, demonstrated here by making use of the Hilbert curve induced orderings. As earlier work indicated, however, reordering of matrix rows and columns also benefits cache use and can limit fill-in. Existing techniques can directly be used in conjunction with vectorised BICRS, potentially increasing performance significantly; similarly, other parallelisation techniques for sparse computations as well as graph computations could benefit from using vectorised BICRS.

If hardware with many cache-coherent cores continue to incur large latencies on out-of-cache accesses, adopting vectorised data structures likely lead to lasting benefits.

To incorporate this vectorised SpMV multiplication in a complete HPC setting, the explicit 2D method from Yzelman and Roose [2] can be used as a top-level method for distributed-memory SpMV multiplication; a BSP version of that algorithm already demonstrated state-of-the-art scalability on highly-NUMA hardware [28].

Acknowledgements

This work was funded by Intel, Janssen Pharmaceutica, and the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). The author thanks the VSC Flemish Supercomputer Center, funded by the Hercules foundation and the Flemish Government department EWI, for facilitating access to its GPU cluster.

References

[1] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi," in *Proc of the 10th Int'l Conf. on Parallel Processing and Applied Mathematics (PPAM)*, 2014, conference, pp. 559–570.

[2] A. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix–vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, 2014.

[3] S. C. Eisenstat, M. Gursky, M. Schultz, and A. Sherman, "Yale sparse matrix package ii: the nonsymmetric codes," DTIC Document, Tech. Rep., 1977.

[4] Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.

[5] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, Eds., *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, PA, 2000.

[6] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[7] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009. [Online]. Available: http://www.math.uu.nl/people/yzelman/publications/yzelman09.pdf

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2009, pp. 233–244.

[9] A. N. Yzelman and R. H. Bisseling, "A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve," in *Progress in Industrial Mathematics at ECMI 2010*, ser. Mathematics in Industry, M. Günther, A. Bartel, M. Brunk, S. Schöps, and M. Striebel, Eds. Berlin, Germany: Springer, 2012, pp. 627–633. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25100-9_73

[10] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," DTIC Document, Tech. Rep., 1993.

[11] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems using ELLPACK*. Springer-Verlag, New York, 1985.

[12] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington, "A sparse matrix library in C++ for high performance architectures," Oak Ridge National Laboratory, Tech. Rep., 1994.

[13] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819108001403

[14] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet another SpMV framework on GPUs," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 107–118. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555255

[15] Y. Nagasaka, A. Nukada, and S. Matsuoka, "Cache-aware sparse matrix formats for kepler GPU," in *20th IEEE International Conference on Parallel and Distributed Computing*, 2014.

[16] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 18.

[17] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *J. Phys. Conf. Series*, vol. 16, pp. 521–530, 2005.

[18] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. 401–423, 2014.

[19] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *Computational Science–ICCS 2005*. Springer, 2005, pp. 99–106.

[20] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. ACM, 1999, p. 30.

[21] A. N. Yzelman and R. H. Bisseling, "Two-dimensional cache-oblivious sparse matrix–vector multiplication," *Parallel Computing*, vol. 37, no. 12, pp. 806 – 819, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001062

[22] R. Ionuţiu, J. Rommes, and W. H. Schilders, "SparseRC: sparsity preserving model reduction for RC circuits with many terminals," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 12, pp. 1828–1841, 2011.

[23] L. Grigori, E. G. Boman, S. Donfack, and T. A. Davis, "Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3426–3446, 2010.

[24] M. Sathe, O. Schenk, B. Uçar, and A. Sameh, "A scalable hybrid linear solver based on combinatorial algorithms," *Combinatorial Scientific Computing*, pp. 95–127, 2012.

[25] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[26] *Intel Xeon Processor E5-1600/E5-2600/E5-4600 Product Families Datasheet - Volume One*, Intel, March 2014, reference number: 329187-003. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-1600-2600-vol-1-datasheet.html

[27] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 273–282.

[28] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen, "MulticoreBSP for C: a high-performance library for shared-memory parallel programming," *International Journal on Parallel Programming*, vol. 42, pp. 619–642, 2014.