CHAPTER 27

SPARSE MATRIX-VECTOR MULTIPLICATION: PARALLELIZATION AND VECTORIZATION

Albert-Jan N. Yzelman, Dirk Roose, Karl Meerbergen
KU Leuven, Belgium

BACKGROUND

Sparse computations are ubiquitous in computational codes, with the sparse matrix-vector (SpMV) multiplication as an important computational kernel in software for simulation (e.g., computational fluid dynamics, structural analysis), optimization (e.g., economics, transport scheduling), data analysis (e.g., drug testing, social networks), and so on. A sparse matrix is characterized by having most of its elements equal to zero. To take advantage of the sparsity, such matrices are stored in specifically designed data structures so that meaningless multiplications with zeroes can be avoided.

Current hardware trends lead to an increasing width of vector units as well as to decreasing effective bandwidth-per-core. For sparse computations, these two trends conflict. In this chapter, we consider sparse matrix computations on multicore architectures with vector processing capabilities, and design a usable and efficient data structure for vectorized sparse computations.

An $m \times n$ matrix A has m rows and n columns and contains elements a_{ij}, with $i = 0,1,\ldots,m-1$ and $j = 0,1,\ldots,n-1$. Consider the matrix-vector multiplication $y = Ax$ with x and y vectors of dimension n and m, respectively. Each element of the output vector y is computed as the dot product of one row of A with the input vector x, i.e., $y_i = \sum_{j=0}^{n-1} a_{ij} x_j$; see Figure 27.1 for an illustration. When all values of A are stored in row-major order such that consecutive elements of a row of A are in consecutive memory locations, then both A and y are read contiguously and with stride-one accesses. Such stream accesses are very efficient and lead to high-bandwidth data movement. Nonobligatory cache misses then only occur on the input vector x, since its elements are read repeatedly.

Taking into account the sizes of the caches, storing A as a collection of blocks that fit precisely into cache can be highly efficient. Such *cache-aware* methods contrast with the development of storage schemes and algorithms that perform well regardless of the specifics of the cache hierarchy. Such *cache-oblivious* methods do not require setting architecture-dependent parameters, such as the size of the aforementioned blocks.

If A is sparse, only the elements $a_{ij} \neq 0$ and their location in the matrix should be stored. These sparse storage schemes are introduced in Section “Sparse matrix data structures.” For operations on compact sparse matrix structures, the number of operations per data word (also known as the arithmetic intensity)

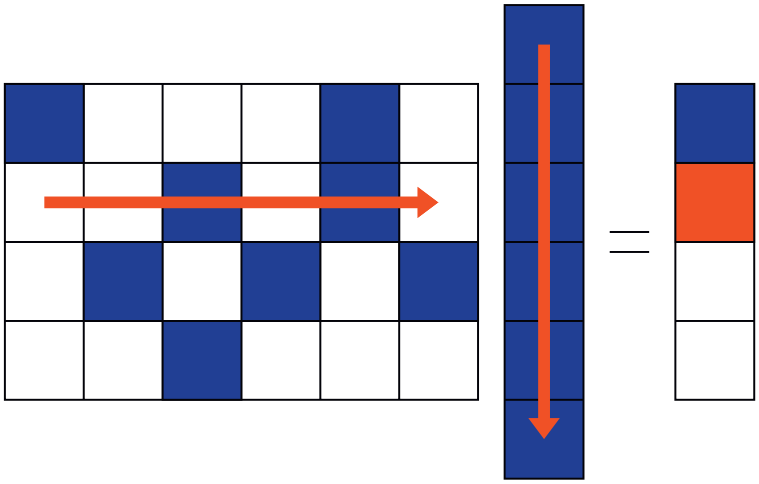

FIGURE 27.1

Illustration of a sparse matrix-vector multiplication $y = Ax$, at y_2. Colored squares represent nonzeroes, while empty squares represent zero values.

is very low; since the time to perform a floating point operation is much shorter than the time to read or store a data word, only a very small fraction of peak performance can be achieved during an SpMV multiplication. This is a characteristic problem for sparse computations and leads to the expectation that

(a) compressing the storage of sparse matrices increases performance and
(b) sparse computations cannot benefit from vectorization.

We show that the former is true even when compression results in additional operations, since the computation remains bandwidth-bound. The latter expectation, however, is *not* always true: vectorization *can* increase the performance of SpMV multiplication on the Intel® Xeon Phi™ coprocessor, despite its low arithmetic intensity.

We describe several sparse matrix data structures in Section "Sparse matrix data structures." Section "Parallel SpMV multiplication" then discusses various strategies to parallelize the SpMV multiplication on a shared memory system, followed by a section that shows how to benefit from vectorization on the Intel Xeon Phi coprocessor. Performance results are given in the final section in this chapter.

SPARSE MATRIX DATA STRUCTURES

A basic data structure for sparse matrix computations is the coordinate (COO) format, which stores a sparse matrix A using three arrays (i,j,v) of length nz each. Here, nz is the number of nonzero elements of A. COO stores the kth nonzero a_{ij} by setting $v_k = a_{ij}, i_k = i,$ and $j_k = j$. Note that the nonzeroes of A can be stored in any order. The following algorithm computes $y = Ax$ using COO.

```
Algorithm 1: COO-BASED SPMV MULTIPLICATION
1: for $k = 0$ to $nz$-1 do
2:     add $v_k \cdot x_{j_k}$ to $y_{i_k}$
3: end for
```

For the COO-based SpMV multiplication, the arithmetic intensity typically lies between 0.25 and 1, depending on the cache efficiency. To make full use of the compute power of current processors, however, a much higher arithmetic intensity is needed.

The Intel Haswell E3-1225 processor, for example, has four 3.2 GHz cores. The processor supports AVX 2 which allows simultaneous (vectorized) operations on four 64-bits data words, in particular also supporting fused multiply-add (FMA) instructions which executes two floating point operations (flops) per data word. The maximum peak performance assumes only FMA operations, multiplied with (1) the number of cores, (2) the number of data words that can be operated simultaneously on (the vector length), and (3) the processor speed: $2\times4\times4\times3.2=102.4$ Gflop/s. The bandwidth of a DDR3-1600 memory controller is 12.8 GB/s, or 1.6 giga-words per second. To make full use of the compute power of this processor, $1024/16=64$ computations must occur for each data word brought to the processor.

The Intel Xeon Phi 7120A coprocessor has 61 cores operating at 1.238 GHz, also supports FMAs, and contains vector units that can handle eight 64-bit data words simultaneously. Its peak performance therefore is $2\times61\times8\times1.238=1208$ Gflop/s. The coprocessor contains 16 GB of GDDR5 memory with a maximum bandwidth of 352 GB/s, or 44 giga-words per second. For algorithms to become compute bound on this processor, an algorithm must perform at least 28 operations per data word ($1208/44=27.5$).

Clearly, for both modern architectures discussed above, the required number of operations per data words lies several factors higher than those incurred for sparse computations such as the SpMV multiplication. Our problem thus is expected to be bandwidth-bound, with cores spending a relatively long time stalling while waiting for required data.

This problem can be alleviated somewhat by inducing more efficient cache behavior. While the access to the sparse matrix data structure itself is contiguous and stride-one, the access patterns of the vectors x and y are determined by the ordering of the nonzeroes: a row-major order induces contiguous stride-one accesses on y but random accesses on x, while a completely random order results in costly random accesses on both x and y. Random accesses (as opposed to contiguous accesses) have high latency cost, lower throughput, and may stall of compute cores. Optimizing the nonzero order such that the memory accesses to x and y are expected to be minimum leads to *fractal storage* schemes; there, space-filling curves impose an ordering on the nonzero coordinates (i,j). Figure 27.2 illustrates a cache-oblivious order of the nonzeroes based on the Hilbert curve.

A *matrix-aware* approach analyzes the nonzero structure of A and optimizes data storage according to this information. Automatic detection of small structured blocks, combined with cache-aware data structures, leads to auto-tuning approaches such as those in the OSKI and pOSKI libraries. For more information on cache- and matrix-aware methods, see, e.g., Vuduc et al. (2005). Alternatively, through sparse matrix partitioning, matrix rows and columns can be reordered such that an upper bound on the number of cache misses is minimized, in an improved cache-oblivious manner. The cost of auto-tuning or preprocessing of matrix-aware methods restricts the usability of this class of methods.

COMPRESSED DATA STRUCTURES

The COO data structure can be compressed in several ways. If we first impose a row major order on the nonzeroes, subsequent entries of i contain strings of the same row number. This redundant data may be compressed into an array s of size $m+1$, where the value for s_i is set to the smallest integer $0 \le k < nz$ for which $i_k = i$, for all $0 \le i < m$; the value of s_m is defined as nz. The data structure (s,j,v), with the latter two arrays unchanged from COO with nonzeroes in row-major order, is the Compressed Row

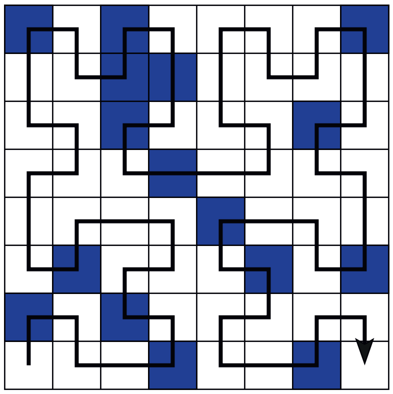

FIGURE 27.2

Traversal of a sparse matrix according to a space-filling Hilbert curve.

Storage (CRS) format. CRS is the *de facto* standard for storing unstructured sparse matrices.[1] Note that using CRS precludes cache-oblivious optimizations by using a fractal nonzero ordering such as in Figure 27.2.

A CRS-based SpMV multiplication loops over all rows first and over its nonzeroes second, as illustrated in the following algorithm:

```
Algorithm 2: CRS-BASED SPMV MULTIPLICATION
1: for $i = 0$ to $m - 1$ do
2:     for $k = s_i$ to $s_{i+1} - 1$ do
3:         add $v_k \cdot x_{j_k}$ to $y_i$
4:     end for
5: end for
```

Similarly, imposing a column-major order and compressing j results in the compressed column storage (CCS). Note that CRS requires $\Theta(2nz + m + 1)$ of storage, which is a substantial reduction compared to $\Theta(3nz)$ for COO.

Compression without limiting the nonzero order to a row- or column-major one is also possible. Consider the COO data structure, with nonzeroes in an arbitrary order, and encode the differences of consecutive values in i and j in the "delta arrays" Δi and Δj, respectively, such that

$$(\Delta i)_k = \begin{cases} i_0 & \text{if } k = 0, \\ i_k - i_{k-1} & \text{if } k > 0. \end{cases}$$

and similarly for Δj. Figure 27.3 (left) illustrates this delta encoding.

When successive nonzeroes belong to the same row or column, strings of zeroes appear in Δi or Δj. This creates an opportunity for compression. Instead of storing Δi, we shall store the array $\widetilde{\Delta i}$, where $(\widetilde{\Delta i})_0 = (\Delta i)_0$ while omitting any remaining zeroes. To not lose information on which nonzero belongs to

[1]CRS is alternatively known as Compressed Sparse Rows (CSR).

$$\begin{pmatrix} & 6 & & & \\ & 7 & & & \\ 2 & & & 4 & \\ & & & & 2 \\ & 3 & 7 & 6 & \end{pmatrix} \rightarrow \begin{array}{rcl} i &=& [\,0\ 3\ 4\ 4\ 4\ 2\ 2\ 1\,] \\ j &=& [\,1\ 4\ 2\ 1\ 3\ 3\ 0\ 1\,] \\ v &=& [\,6\ 2\ 7\ 3\ 6\ 4\ 2\ 7\,] \end{array}$$

$$\swarrow$$

$$\begin{array}{rcl} \Delta i &=& [\,0\ 3\ 1\ 0\ 0\ {-2}\ 0\ {-1}\,] \\ \Delta j &=& [\,1\ 3\ {-2}\ {-1}\ 2\ 0\ {-3}\ 1\,] \\ \mathrm{v} &=& [\,6\ 2\ 7\ 3\ 6\ 4\ 2\ 7\,] \end{array} \rightarrow \begin{array}{rcl} \widetilde{\Delta i} &=& [\,0\ 3\ 1\ {-2}\ {-1}\,] \\ \widetilde{\Delta j} &=& [\,1\ \mathbf{8}\ \mathbf{3}\ {-1}\ 2\ \mathbf{5}\ {-3}\ \mathbf{6}\,] \\ \mathrm{v} &=& [\,6\ 2\ 7\ 3\ 6\ 4\ 2\ 7\,] \end{array}$$

FIGURE 27.3

Sketch of an example matrix (top left) and a COO representation of it (top right). By omitting zeroes from Δi of the COO delta encoding (bottom left), the BICRS data structure (bottom right) can be derived. A bold-faced element in $\widetilde{\Delta j}$ indicates that a row jump occurs at the corresponding nonzero; note that this means n was added to the corresponding column increment.

which row, we adapt the array Δj so that it marks whenever a row jump in Δi occurred, by adding n to the corresponding element in the column increment array. The resulting array $\widetilde{\Delta j}$ thus is determined by

$$\widetilde{\Delta j}_k = \begin{cases} \Delta j_0 & \text{if } k = 0, \\ \Delta j_k & \text{if } k > 0 \text{ and } \Delta i_k = 0, \\ \Delta j_k + n & \text{if } k > 0 \text{ and } \Delta i_k \neq 0. \end{cases}$$

Figure 27.3 (right) illustrates this approach. Note that the additional bookkeeping in the column increment array essentially costs a single bit per entry.

Similar to CRS and CCS, the roles of rows and columns could be interchanged with compression applied on Δj instead. These data structures are called Bi-directional Incremental CRS (BICRS) and BICCS, respectively. BICRS allows for efficient implementations, as shown in Algorithm 3. BICRS was introduced to make the use of space-filling curves feasible in terms of data movement. See Yzelman and Bisseling (2012) for details.

```
Algorithm 3: BICRS-BASED SPMV MULTIPLICATION
1: $c = 0, i = \tilde{\Delta} i_c, j = \tilde{\Delta} j_0$
2: **for** $k = 0$ to $nz$-1 **do**
3:     add $v_k \cdot x_j$ to $y_i$
4:     add $\tilde{\Delta} j_{k+1}$ to $j$
5:     **if** $j$ indicates a row jump **then**
6:         add $\tilde{\Delta} i_c$ to $i$
7:         increment $c$
8:     **end if**
9: **end for**
```

Note that only $\Theta(\log_2 m)$ and $\Theta(\log_2 n)$ bits are required to store an element of $\widetilde{\Delta i}$ and $\widetilde{\Delta j}$, respectively. By using shorter integer types for storing these arrays, additional compression is achieved; we refer to this data structure as compressed BICRS.

BLOCKING

Using Compressed BICRS is synergistic with blocking of the sparse input matrix. We use a two-level approach where A is subdivided into small square blocks, and the blocks are ordered according to a space-filling Hilbert curve. Ordering the blocks this way increases cache efficiency, and can be efficiently stored using BICRS. Construction of this higher-level data structure is cheap since Hilbert COOs must now be calculated for each block, instead of for each nonzero, which also applies to the sorting of blocks (instead of nonzeroes) according to their Hilbert coordinates.

Within each block, nonzeroes are stored in row-major order. We optimize the dimensions of each block so that the elements of $\widetilde{\Delta i}$ and $\widetilde{\Delta j}$ can be stored in 16 bits (using `short int`). This enables the compressed storage of nonzeroes within blocks using Compressed BICRS; since most rows within sparse blocks are empty, BICRS is preferable to CRS since the latter requires $m+1$ storage regardless of whether any of those rows are empty. By using compression via shorter integer types, this two-level scheme ensures that storing the blocked matrix requires less memory than storing the original unblocked matrix A using CRS. See Yzelman and Roose (2014) for more details.

PARALLEL SPMV MULTIPLICATION

To parallelize the algorithms above for shared-memory architectures, the work must be distributed over the available threads or processes. In general, data reside in the global memory, accessible by all threads. If the matrix and vectors are allocated in one contiguous chunk of memory, then each thread can access elements within that chunk according to the given work distribution. This way, the CRS-based SpMV multiplication can easily be parallelized using OpenMP compiler directives by adding the following pragma before the outer for-loop (statement 1) in Algorithm 2: `omp parallel for schedule(dynamic,8)`.

However, data can also be distributed explicitly by having each core allocates their own memory chunks, so that each thread allocates the data elements it operates during the parallel computation. There are two reasons for using explicit data distributions on shared-memory architectures: (1) to avoid data races and (2) to exploit data locality. The choice of distributing data explicitly or keeping all data globally available can significantly impact the performance. In the below, we describe two types of data distributions for the SpMV multiplication that apply explicit distributions in varying degrees.

PARTIALLY DISTRIBUTED PARALLEL SPMV

In the partially distributed approach, illustrated in Figure 27.4, the matrix A is partitioned row-wise, yielding submatrices A_s, $s=0,\dots,p-1$, with p the number of threads. Each submatrix A_s is a rectangular matrix with less than m rows but with the full number of columns n. Each is stored in separate chunks of memory, conform the idea of explicit distributions. To ensure work balance, each A_s should contain approximately nz/p elements (although the number of rows may differ from m/p). Each A_s is divided into blocks with fixed row and column dimensions and uses (compressed) BICRS storage, as described in the earlier section on blocking. The blocks are handled in an order defined by the Hilbert curve. This ensures a cache-oblivious traversal that benefits data reuse on the high-level caches, while minimizing the amount of memory required for data storage.

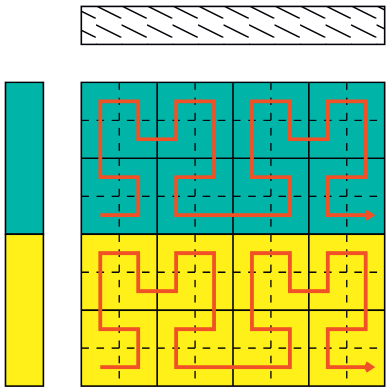

FIGURE 27.4

Illustration of the partially distributed method for $p=2$. The upper and lower matrix regions are distributed to different threads. The distribution ensures each thread handles approximately the same number of nonzero elements; thus, in practice, the height of each region may differ. The output vector (left) is distributed according to the row distribution. The input vector (top) is not distributed explicitly. The Hilbert curve in the upper and lower regions indicates the cache-oblivious access pattern of the matrix blocks.

In accordance to the distribution of the matrix rows, the output vector y is also split in p contiguous and non-overlapping blocks, with the sth block allocated by thread s itself. Threads access only the y_s they allocated themselves, thus avoiding data races (concurrent writes) on the output vector, while simultaneously exploiting data locality on y. All threads still operate on the entire input vector x, which is stored in an interleaved fashion across the multiple memory banks that may be available. This happens automatically on the Intel Xeon Phi coprocessor, but requires manual intervention via the “libnuma” library on other shared-memory architectures. This prevents exploiting any data locality on x. By nature of this row-wise distribution and the global availability of the vector x, no explicit interthread communication or synchronization is required during SpMV multiplication. For a more detailed description, see Yzelman and Roose (2014).

The following algorithm sketches this approach. Note that the last line (statement 4) is optional, and only required when the application cannot work with a distributed output vector. To exploit data locality and for increased cache efficiency, however, efficient applications should phrase operations on y as thread-local operations on each y_s separately.

```
Algorithm 4: PARTIALLY DISTRIBUTED PARALLEL SPMV MULTIPLICATION
1: Partition A row-wise into local matrices $A_s, s=0,\ldots,p-1$ (see Fig. 27.4)
2: Create corresponding local arrays $y_s, s=0,\ldots,p-1$
3: Each core $s=0,\ldots,p-1$ executes SpMV$(A_s, x, y_s)$
4: Concatenate $y_s, s=0,\ldots,p-1$ into $y$
```

FULLY DISTRIBUTED PARALLEL SPMV

In a fully distributed scheme, we exploit data locality not only on A and y but also on x. In a preprocessing step, we first partition the sparse matrix by splitting the nonzeroes of A into p parts. These again form local matrices A_s, on which each thread can concurrently perform local SpMV multiplications $y_s = A_s x_s$. In this case, however, rows and columns of the A_s may overlap; the x_s and y_s thus are possibly overlapping subsets of the input and output vectors. A good matrix partitioning keeps the number of nonzeroes of A_s close to nz/p and minimizes the communication involving the overlapping parts of A_s, x_s, and y_s.

Figure 27.5 shows the partitioning of a matrix in the doubly separated block diagonal (SBD) form, as introduced by Yzelman and Bisseling (2009, 2011), for four threads. Such a partitioning can be obtained by (recursive) hypergraph bi-partitioning, which usually entails preprocessing the input matrix using a matrix partitioner. For examples of such partitioners, see Mondriaan by Vastenhouw and Bisseling (2005) or Zoltan by Devine et al. (2006). The partitioning also defines the reordering required to permute A into SBD form. This reordering results in p large local blocks (the squares in the figure), separated by $p-1$ separator crosses. Thread-local multiplication of the local blocks requires no communication, while multiplications involving nonzeroes in separator crosses requires explicit inter-thread communication, and also incurs synchronization overheads to prevent data races on the output vector elements corresponding to matrix rows contained in the separator cross.

The thread-local multiplications do not use the explicit blocking scheme described in the earlier section on blocking, but instead rely on the blocks naturally created by the doubly SBD reordering. These local blocks and separator crosses are stored in separate Compressed BICRS data structures, for which the data types of the $\widetilde{\Delta i}$ and $\widetilde{\Delta j}$ arrays are auto-tuned at run time for maximum compression.

Explicitly distributing all data is mandatory on distributed-memory architectures: modern-day supercomputing requires fully distributed algorithms to parallelize over multiple nodes, while other methods may be used within nodes. The fully distributed SpMV multiplication method sketched here is efficient on both distributed-memory as well as shared-memory architectures, as demonstrated recently by Yzelman et al. (2014); when used on shared-memory multisocket machines, it is capable of

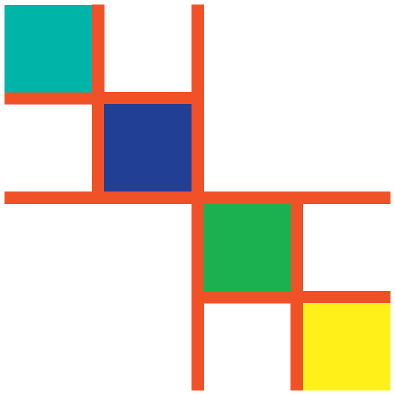

FIGURE 27.5

Illustration of the fully distributed and reordered SpMV multiplication for four threads. Each of the square blocks correspond to submatrices distributed to different threads. Between the square blocks separator crosses appear. Nonzeroes in separator crosses may be distributed to any thread the separator spans. Whenever an element of x overlaps horizontally with a separator cross, multiple threads may read that element; likewise, an element from y may be written to by multiple threads if its location overlaps with the vertical part of a separator cross. In practice, the blocks are usually not of equal size.

outperforming the partially distributed parallel SpMV multiplication described earlier in this section, as well as other state-of-the-art methods.

Since the cost of matrix partitioning is non-trivial and increases with the number of threads, this method will not be applied on the Intel Xeon Phi coprocessor. However, when work is to be distributed over multiple Intel Xeon Phi coprocessors, as in a classical distributed-memory supercomputing context, fully distributed methods are mandatory for good performance. For more detail on this method, refer to Yzelman and Roose (2014). For a high-performance implementation of this scheme, refer to its MulticoreBSP for C implementation by Yzelman et al. (2014).

VECTORIZATION ON THE INTEL XEON PHI COPROCESSOR

The vector units on the Intel Xeon Phi coprocessor operate on vector registers that each contain eight 64-bit floating point values. The low flop-to-byte ratio of the SpMV multiplication suggests that these vector units will not be usable for a bandwidth-bound computation; the computation is memory-bound, with processing units more often waiting for data to arrive than performing computations on said data. However, simply using the partially distributed parallel SpMV described in the previous section with an increasing number of threads p, up to $p=240$, reveals an effective bandwidth use far less than the available 352 GB/s. This indicates that the SpMV on the coprocessor is not bandwidth-bound (nor did it turn compute bound). Instead, the multiplication on this architecture has become latency bound; using the full amount of 240 threads does not saturate the memory subsystem. A way to still be able to generate more data requests while using the same number of threads is to use vectorization.

Auto vectorization of a sequential SpMV multiplication using Compressed BICRS is, however, not possible. Pointer arithmetic and indirect addressing of vector elements prevents this. To enable vectorization, the BICRS multiplication algorithm is rewritten to operate on successive chunks of $p{\cdot}q$ nonzeroes, where $p{\cdot}q = l$ is the vectorization length, so that vector registers can operate op l nonzeroes simultaneously.

We refer to the different possible choices of p and q as the *blocking size*; on the coprocessor, $p{\cdot}q = 8$ and four possible blocking sizes thus appear: 1×8, 2×4, 4×2, and 8×1. These are illustrated in Figure 27.6. If $\mathbf{y}$, $\mathbf{v}$, and $\mathbf{x}$ are arrays of length 8 that correspond to vector registers, then the inner computation of the SpMV multiply, i.e., $y_i = a_{ij}x_j$, can instead be written as a single vectorized FMA $\mathbf{y} := \mathbf{y} + \mathbf{v} * \mathbf{x}$. Rewriting the BICRS SpMV multiplication algorithm to use this type of vectorization requires the use of gather and scatter instructions on the coprocessor; the gather allows loading non-consecutive

$$\begin{array}{c|cccccccc} 1\times 8 & j_0 & j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 \\ \hline i_0 & v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 \end{array} \qquad \begin{array}{c|cccccccc} 2\times 4 & j_0 & j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 \\ \hline i_0 & v_0 & v_1 & v_2 & v_3 & & & & \\ i_1 & & & & & v_4 & v_5 & v_6 & v_7 \end{array}$$

$$\begin{array}{c|cccccccc} 4\times 2 & j_0 & j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 \\ \hline i_0 & v_0 & v_1 & & & & & & \\ i_1 & & & v_2 & v_3 & & & & \\ i_2 & & & & & v_4 & v_5 & & \\ i_3 & & & & & & & v_6 & v_7 \end{array} \qquad \begin{array}{c|cccc} 8\times 1 & j_0 & j_1 & \cdots & j_7 \\ \hline i_0 & v_0 & & & \\ i_1 & & v_1 & & \\ \vdots & & & \ddots & \\ i_7 & & & & v_7 \end{array}$$

FIGURE 27.6

The four possible blocking sizes on the coprocessor: 1×8, 2×4, 4×2, and 8×1.

elements of an array into a vector register, while the scatter does the inverse. Using these primitives, the following pseudocode illustrates vectorized BICRS multiplication.

```
1: while there are nonzeroes remaining do
2:     get the next set of $p$ rows to operate on
3:     gather the corresponding output vector elements in $\mathbf{y}$
4:     while there are nonzeroes remaining on any of the current rows do
5:         get the next set of $p \cdot q$ (possibly overlapping) column indices
6:         gather the corresponding input vector elements in $\mathbf{x}$
7:         retrieve the next set of $p \cdot q$ nonzero values $\mathbf{v}$
8:         do the vectorized multiply-add $\mathbf{y} := \mathbf{y} + \mathbf{v} \cdot \mathbf{x}$
9:     end while
10:    scatter the cached output vector elements $\mathbf{y}$ back to main memory.
11: end while
```

Upon processing all nonzeroes on each of the p rows, the l entries of **y** need to be reduced by summation into p entries; e.g., for 2×4 blocking, the top half of **y** corresponds to i_0 in Figure 27.6, while the bottom half corresponds to i_1. The 1×8 case thus requires an allreduce into i_0, while the 8×1 blocking requires no reduction at all. Using other block sizes leads to partial reductions.

This algorithm requires a slight modification of the BICRS data structures as well; for each $p \times q$ block, the indices of the last $p \cdot q - 1$ nonzeroes are relative to the first element of that block. Figure 27.7

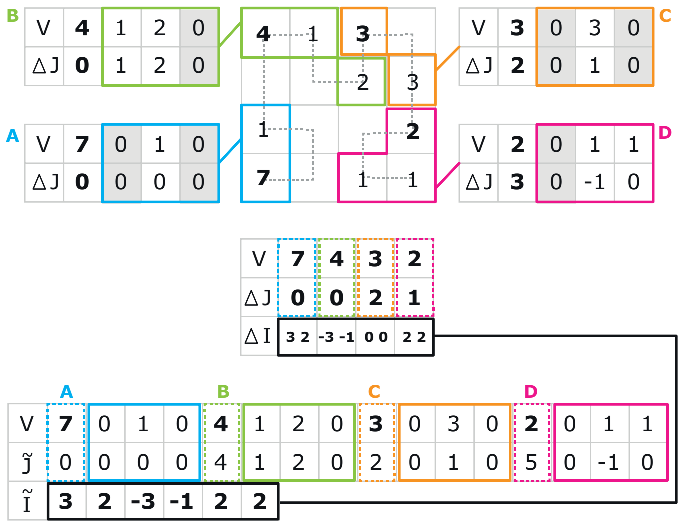

FIGURE 27.7

Illustration of the final vectorized BICRS data structure. It separates the relative encoding of $pq - 1$ nonzeroes in each block from the BICRS encoding of each of the leading nonzeroes of all blocks; these two encodings are then combined in the final vectorized data structure. The figure illustrates 2×2 blocking on an 4×4 matrix with 10 nonzeroes ordered according to a Hilbert curve. This results in 4 blocks, containing 6 explicit zeroes (fill-in).

illustrates this principle. When there are not enough nonzeroes in A to fill a block corresponding to a fixed set of rows, then explicit zeroes have to be stored; this *fill-in* results in additional storage and explicit multiplications using zeroes, an overhead which is hoped to be offset by the gain in effective bandwidth use.

IMPLEMENTATION OF THE VECTORIZED SPMV KERNEL

Figures 27.8 and 27.9 provide a generic C++ implementation of the vectorized BICRS SpMV multiplication. In these code snippets, the variables x,y,l,p,q are as defined in the text. The template variables _i_type and _v_type correspond to the index types and value types, respectively. Pointers to the arrays $\widetilde{\Delta i}$ and $\widetilde{\Delta j}$ are named row_index_array and col_index_array; a pointer to v is denoted by *value_array*. All other variables are declared within the code snippets. The code shown in the figures is functional and is directly derived from the production code distributed with the book.[2]

Specializing code for specific architectures and block sizes can lead to better performance than a compiler-optimized generic implementation would achieve. For cases where p is very small, for instance, it may not make sense to use gather and scatter primitives on the output vector. Figure 27.10 shows a specialized implementation for handling 1×8 blocks on the Intel Xeon Phi, optimized using

```
void spmv( const double *__restrict__ x, double *__restrict__ y ) {
    //declare buffers
    __declspec(align(32)) _i_type c_ind_buffer[ l ];
    __declspec(align(32)) _i_type r_ind_buffer[ l ];
    __declspec(align(32)) _v_type input_buffer[ l ];
    __declspec(align(32)) _v_type outputbuffer[ l ];
    __declspec(align(32)) _v_type outputvector[ l ];

    //shift input vector, output vector to first nonzero
    x += *col_index_array; y += *row_index_array;

    //fill buffers and load first l output elements
    for( size_t i = 0; i < l; ++i ) {
        c_ind_buffer[ i ] = *col_index_array++;
        r_ind_buffer[ i ] = *row_index_array++;
        outputbuffer[ i ] = 0;
        outputvector[ i ] = y[ r_ind_buffer[ i ] ];
    }

    //reset start column index, start row index
    c_ind_buffer[ 0 ] = 0; r_ind_buffer[ 0 ] = 0;

    //keep track of how many row blocks in outputvector were processed
    size_t blockrow = 0;
    ...
```

FIGURE 27.8

C++ implementation of the vectorized BICRS SpMV multiplication for generic architectures, data types, and block sizes. This is the kernel initialization code. The code listing continues in Figure 27.9.

[2]The latest code version is available from http://albert-jan.yzelman.net/software#SL.

```
...
//start kernel
while( value_array < value_array_end ) {
    //process row
    while( x < x_end ) {
        //fill input buffer (gather)
        for( size_t i = 0; i < l; ++i ) input_buffer[ i ] = x[ c_ind_buffer[ i ] ];

        //do FMA
        for( size_t i = 0; i < l; ++i ) outputbuffer[ i ] += value_array[ i ] * input_buffer[ i ];

        //shift nonzero vector
        value_array += l;

        //shift input vector
        x += *col_index_array;

        //fill c_ind_buffer
        for( size_t i = 0; i < l; ++i ) c_ind_buffer[ i ] = *col_index_array++;
        c_ind_buffer[ 0 ] = 0;
    }

    //reduce l outputbuffer elements to p row contributions
    for( size_t i = 0; i < p; ++i )
        for( size_t j = 0; j < q; ++j )
            outputvector[ p*blockrow + i ] += outputbuffer[ i*q + j ];

    //prepare for next block of p rows
    ++blockrow;
    for( size_t i = 0; i < l; ++i ) outputbuffer[ i ] = 0;

    //undo row change signal, shift input vector
    x -= n;

    //if all elements of outputvector were updated
    if( blockrow == q ) {

        //write back outputvector
        for( size_t i = 0; i < l; ++i ) y[ r_ind_buffer[ i ] ] = outputvector[ i ];

        //shift output vector to new row
        y += *row_index_array;

        //load new r_ind_buffer
        for( size_t i = 0; i < l; ++i ) r_ind_buffer[ i ] = *row_index_array++;
        r_ind_buffer[ 0 ] = 0;

        //read new outputvector
        for( size_t i = 0; i < l; ++i ) outputvector[ i ] = y[ r_ind_buffer[ i ] ];

        //reset block row counter
        blockrow = 0;
    }
    //write back any modified items outputvector may hold
    for( size_t i = 0; i < l; ++i ) y[ r_ind_buffer[ i ] ] = outputvector[ i ];
}
}
```

FIGURE 27.9

C++ implementation of the inner kernel code of the vectorized BICRS SpMV multiplication for generic architectures, data types, and block sizes. The code was initialized in Figure 27.8.

```
void spmv( const double *__restrict__ x, double *__restrict__ y ) {
    __m512d input_buffer, value_buffer, outputbuffer;
    __m512i c_ind_buffer, zeroF;
           zeroF = _mm512_set_epi32( 1, 1, 1, 1, 1, 1, 1, 0 );
    outputbuffer = _mm512_setzero_pd();

    //load in column indices of the first block, and set c_ind_buffer[0]=0
    c_ind_buffer = _mm512_load_epi32 ( col_index_array );
    c_ind_buffer = _mm512_mullo_epi32( c_ind_buffer, zeroF );

    //shift vector pointers to the first nonzero position
    x += *col_index_array; col_index_array += l;
    y += *row_index_array++;

    //start kernel
    while( value_array < value_array_end ) {
        //process current row
        while( x < x_end ) {
            //gather input vector elements
            input_buffer = _mm512_i32logather_pd( c_ind_buffer, x, 8 );

            //load (stream) nonzero values
            value_buffer = _mm512_load_pd( value_array ); value_array += l;

            //do FMA; outputbuffer += value_buffer * input_buffer
            outputbuffer = _mm512_fmadd_pd( value_buffer, input_buffer, outputbuffer );

            //load in next block, shift input vector
            c_ind_buffer = _mm512_load_epi32 ( col_index_array );
            c_ind_buffer = _mm512_mullo_epi32( c_ind_buffer, zeroF );
            x += *col_index_array; col_index_array += l;
        }
        //write out local contributions (via allreduce), and reset
                  *y += _mm512_reduce_add_pd( outputbuffer );
        outputbuffer = _mm512_setzero_pd();

        //shift input vector back to a valid position
        x-= n;

        //shift output vector to next row position
        y += *row_index_array++;
    }
}
```

FIGURE 27.10

C++ implementation of the vectorized BICRS SpMV multiplication for the Intel Xeon Phi using ICC intrinsics. Assumes 64-bit floats, 32-bit indices, and 1×8 blocks.

ICC intrinsics for the AVX-512 instruction set. It assumes 32-bit index types (_i_type=**int32_t**), 64-bit floating point values (_v_type=double), and performs write-backs to *y* using scalar FMAs.

The Xeon Phi specific code, distributed with the book, employs shorter index types (_i_type=**int16_t**) to exploit the compression opportunities sparse blocking makes possible. A single load of c_ind_buffer thus reads in 16 integers instead of 8, since the 512-bit registers can contain 512/16 = 16 short ints. This requires the inner while-loop in Figure 27.10 to be unrolled in two parts, where the first part uses the upper half of c_ind_buffer. This is followed by code that handles a possible row jump, after which the upper half of c_ind_buffer is swapped with the lower half (which contains the remaining unhandled column increments), after which the inner kernel code is repeated. After the manual unroll ends, the loop continues and the next 16 column increments are read in. Such unrolls are also necessary when using gather and scatter instructions on the output vector while $p < l$: partial reductions needed when $1 < p < l$ require additional permutations and additions involving outputbuffer, while adding the thus obtained p contributions to outputvector requires masked vector additions that require unrolling for efficiency. For brevity, these codes are not included here; please refer to the distributed code to view the unrolled codes in detail.

EVALUATION

We have compared the execution speed measured in Gflop/s of three parallel SpMV algorithms, namely, (1) the OpenMP-enabled CRS SpMV multiplication, (2) the partially distributed parallel SpMV method (with local matrices stored using compressed BICRS) implemented using PThreads, and (3) a vectorized version of the partially distributed method (using the vectorized BICRS data structure as presented in the previous section), also implemented using PThreads. We compare the results to the CRS-based implementation from the Intel Math Kernel Library version 11.1.1. All code was compiled using the Intel C/C++ Compiler (ICC) version 14.0.1.

We ran tests using six matrices of various dimensions and sparsity patterns (see Figure 27.11). The selected matrices are representative of four generic classes: they are either small (so that the corresponding vectors fit into the combined caches) or large, and structured (so that the natural nonzero structure benefits cache reuse) or unstructured. See Yzelman and Roose (2014) for an application of this categorization on a much wider set of real-world matrices.

Small matrices	Rows	Columns	Nonzeroes	
nd24k	72,000	72,000	28,715,634	U
s3dkt3m2	90,449	90,449	1,921,955	S
Large matrices				
Freescale1	3,428,755	3,428,755	17,052,626	S
wiki07	3,566,907	3,566,907	45,030,389	U
cage15	5,154,859	5,154,859	99,199,551	S
adaptive	6,815,744	6,815,744	13,624,320	U

FIGURE 27.11

Matrices used in the experiments. The matrices above the horizontal separator line are considered small; their corresponding input and output vectors have sufficiently small combined dimensions to fit into most (combined) L2 caches of modern architectures. The right column indicates whether the sparse matrix has a beneficial structure (S for structured) with respect to cache reuse during a CRS-based SpMV multiplication, or whether the matrix has no such structure (U for unstructured).

ON THE INTEL XEON PHI COPROCESSOR

We ran our experiments on an Intel Xeon Phi 7120A coprocessor, which contains 61 cores at 1.238 GHz and 16 GB of memory. Each core comes equipped with a 32 kB L1 cache and a 256 kB L2 cache, supporting four hardware threads per core. We used all threads on all cores, except for the core reserved for the operating system, thus employing 240 threads.

Figure 27.12 shows the performance of the parallel SpMV multiplication using the partially distributed method with vectorized BICRS. We ran experiments on each matrix using all of the possible blocking sizes (1×8, 2×4, 4×2, or 8×1). These choices lead to different amounts of fill-in, which are reported in Figure 27.13. Comparing the results, choosing the block size that result in the least amount of fill-in almost always corresponds to the fastest execution among the possible block sizes; the only exception was on the matrix wiki07, where vectorization had an adverse effect compared to nonvectorized SpMV multiplication. The results indicate that vectorization is highly effective on the Intel Xeon Phi coprocessor, especially for small matrices (such as nd24k and s3dkt3m2), where thread-local parts of the vectors can reside in fast local caches.

The performance figures reported in Figure 27.17 exclude the time required for preprocessing. For the partially distributed methods, this overhead is negligible compared to the construction of a CRS data structure, since the latter requires sorting of all nonzeroes. Determining the fill-in for all possible block sizes *a priori* can be done by a single traversal of the input matrix before the actual matrix construction.

To assess the performance improvement of the various optimizations discussed above, we measured the performance of the OpenMP CRS implementation, successively augmented with

1. partial distribution of A and x,
2. sparse blocking with the Hilbert cache-oblivious order on the blocks, and
3. vectorized BICRS storage of the local A.

These results are reported in Figure 27.14.

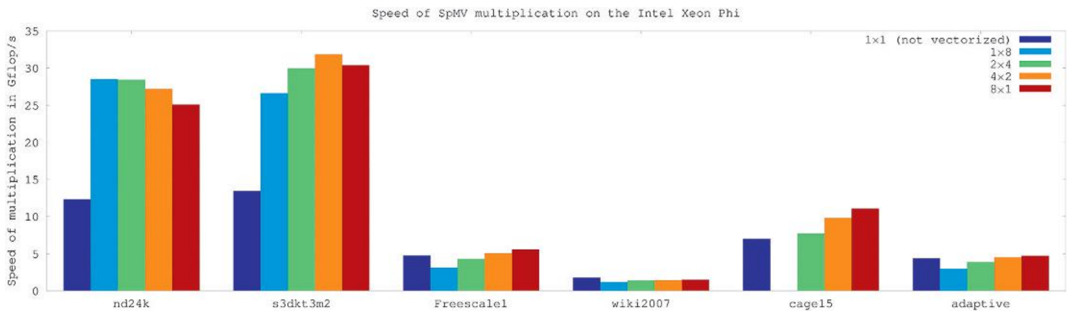

FIGURE 27.12

Performance of the partially distributed method using vectorized BICRS on each of the four possible blocking sizes 1×8, 2×4, 4×2, and 8×1. For reference, the results corresponding to the non-vectorized partially distributed method are included under the 1×1 block size. A value is missing for cage15 with 1×8 blocking; in this case, the extra amount of fill-in caused the coprocessor to run out of memory before the experiment could successfully complete.

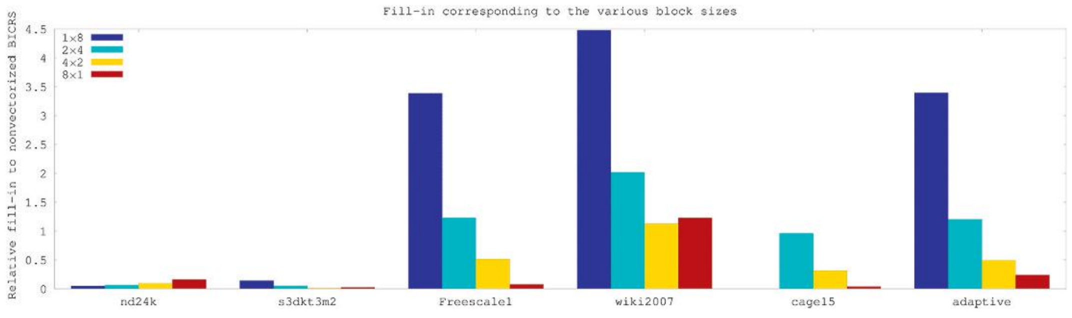

FIGURE 27.13

Relative fill-in corresponding to the matrices and block sizes in Figure 27.12. The reported figure is equal to the number of explicitly added zeroes divided by the number of nonzeroes contained in the matrix original. A value for the cage15 and 1×8 blocking is missing since the added fill-in (of presumably a factor 2) caused the coprocessor to run out of memory.

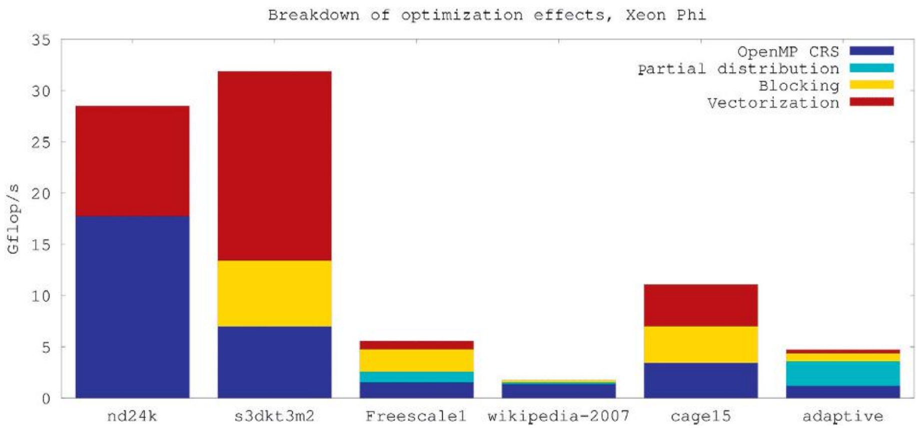

FIGURE 27.14

Performance improvement due to the various optimizations discussed in this chapter, on the Intel Xeon Phi. The performance baseline is OpenMP CRS, successively augmented with (1) partial data distribution, (2) sparse blocking with Hilbert ordering, and (3) vectorized BICRS.

Which optimizations are most effective strongly depends on both the matrix dimensions as well as its type. On the nd24k matrix, for example, partial data distributions and blocking do not improve the OpenMP CRS baseline performance, while vectorization speeds up its performance by about 50%. In contrast, the major performance increase on the adaptive matrix is due to partially distributing the data, while for the cage15 matrix, the main performance increase comes equally from blocking with Hilbert ordering as well as vectorization.

ON INTEL XEON CPUs

Another test platform is a dual-socket CPU-based machine, containing two Intel® Xeon® E5-2690 v2 "Ivy Bridge EP" processors. Each processor contains 10 cores equipped with 32 kB L1 and 256 kB L2 local caches, while the 10 cores share a 25 MB L3 cache. Both sockets connect to local memory banks containing 64 GB of DDR3 memory running at 1600 MHz, thus achieving a maximum bandwidth of two times 12.8 GB/s.

Ivy Bridge processors support vector instructions on 256-bit registers, hence $l = 4$ when using double precision floating point data. AVX does not support gather and scatter instructions, however, so the vectorized scheme on this architecture will likely be less effective than on the Intel Xeon Phi. Gather instructions are available in AVX2-enabled processors (the “Haswell” line), while scatter instructions are planned for AVX3.

Experiments were run using 40 threads, thus using the hyperthreading capabilities available on this processor. Figure 27.15 shows the performance of the vectorized SpMV multiplication on the dual-socket machine. Figure 27.16 illustrates the effect of the various optimizations on this architecture.

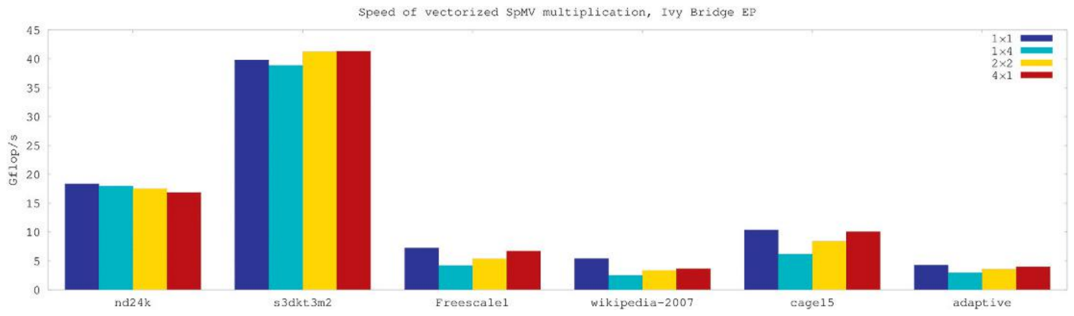

FIGURE 27.15

Performance of the partially distributed method using vectorized BICRS on a dual-socket Intel “Ivy Bridge EP.” The performance of each possible blocking size (1 × 4, 2 × 2, and 4 × 1) is reported and compared against results obtained using nonvectorized BICRS (1 × 1).

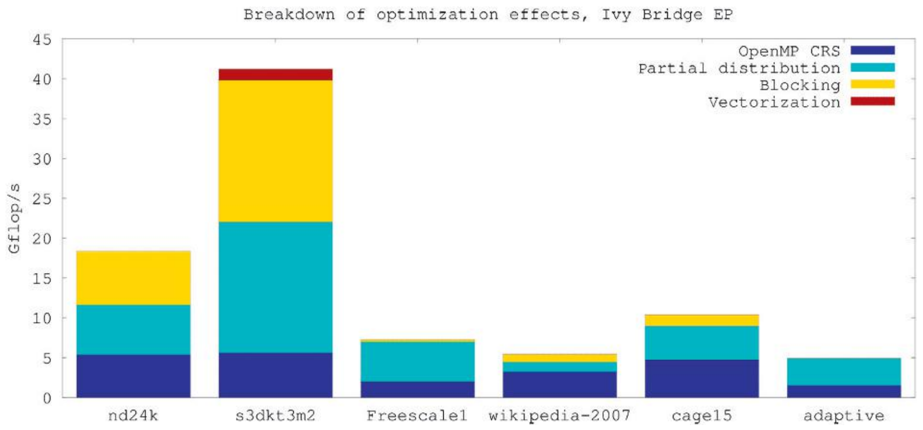

FIGURE 27.16

Performance improvement due to the various optimizations discussed in this chapter, on a dual-socket Intel “Ivy Bridge EP.” The performance baseline is OpenMP CRS, successively augmented with (1) partial data distribution, (2) sparse blocking with Hilbert ordering, and (3) vectorized BICRS.

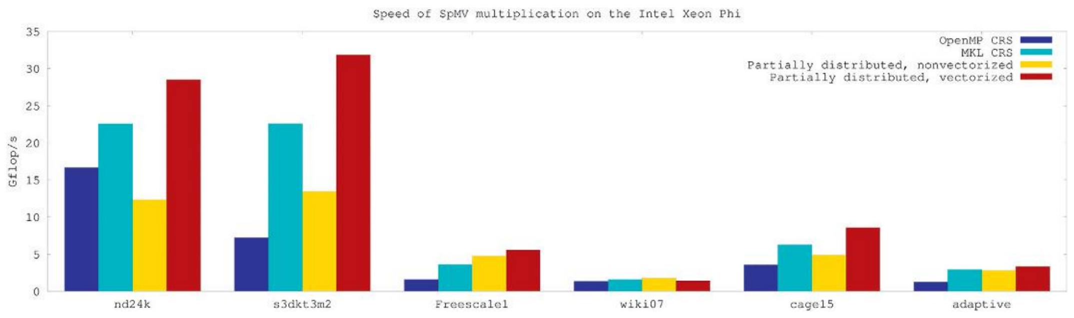

FIGURE 27.17

Performance of four implementations of a parallel SpMV multiplication using 60 cores (240 threads) on the Intel Xeon Phi 7120A coprocessor: (a) OpenMP-enabled parallel CRS, (b) MKL-enabled parallel CRS, (c) partially distributed method using compressed BICRS, and (d) partially distributed method using vectorized BICRS.

Similar to the results on the Intel Xeon Phi, the effect of each optimization largely depends on the size and type of matrix. In general, however, the effects of vectorization are noticeably less pronounced than on the coprocessor. This is due to the absence of hardware support for the gather and scatter instructions, and the fact that the SpMV multiplications are bandwidth-bound on CPUs (and not latency bound as on the coprocessor).

PERFORMANCE COMPARISON

Figure 27.17 compares the performance of the vectorized SpMV on the Xeon Phi to the following alternative methods: (1) the straightforward OpenMP CRS solution, (2) the partially distributed solution using nonvectorized BICRS, and (3) the SpMV multiplication code included with Intel MKL. For the larger matrices in our test set, the nonvectorized partially distributed algorithms and the MKL SpMV codes display a significantly better performance compared to the simple OpenMP-parallelized CRS algorithm. The performance of the partially distributed algorithm significantly increases with the introduction of vectorization, and outperforms the CRS SpMV multiplication from Intel MKL, with the exception of the unstructured wiki07 matrix.

Figure 27.18 makes a similar comparison for the dual-socket "Ivy Bridge EP" machine. As vectorization did not always lead to better performance on this architecture, the nonvectorized partially distributed method outperforms all other alternatives, except on the structured s3dkt3m2 matrix (where vectorization did result in increased performance).

SUMMARY

The difficulties in achieving a high performance for SpMV multiplication lie with a low flop-to-byte ratio and inefficient cache use. In the case of the Intel Xeon Phi coprocessor, an additional difficulty is the high-latency memory access.

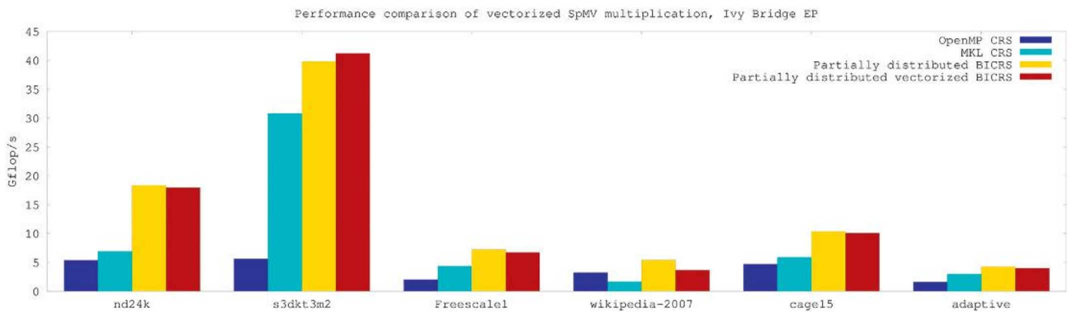

FIGURE 27.18

Performance of parallel SpMV implementations on 40 threads on a dual-socket Ivy Bridge machine. (a) OpenMP CRS, (b) MKL CRS, (c) partially distributed BICRS, and (d) partially distributed vectorized BICRS.

In this chapter, we discussed strategies that maximize data locality and data reuse for the parallel SpMV multiplication on shared-memory systems. We focus on data structures that support vectorized operations, in addition to arbitrary nonzero traversals for cache-obliviousness and advanced compression through blocking. Parallelization schemes for both shared-memory and distributed-memory platforms were discussed.

Experiments were performed on both the Intel Xeon Phi coprocessor and on a dual-socket Ivy Bridge shared-memory system. The results show that the shared-memory parallelization technique, in conjunction with sparse blocking, cache-oblivious traversals, compression, and vectorization, outperforms not only a CRS-based algorithm implemented using OpenMP, but also the SpMV-code available in the Intel MKL library; optimizing for the same programming model thus leads to efficient codes on both architectures.

ACKNOWLEDGMENTS

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

FOR MORE INFORMATION

Sparse Library: http://albert-jan.yzelman.net/software#SL.
MulticoreBSP for C: http://www.multicorebsp.com.

Devine, K.D., Boman, E.G., Heaphy, R.T., Bisseling, R.H., Catalyurek, U.V., 2006. Parallel hypergraph partitioning for scientific computing. In: Proceedings IEEE International Parallel and Distributed Processing Symposium 2006. IEEE Press, Long Beach, CA.
Vastenhouw, B., Bisseling, R.H., 2005. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. 47 (1), 67–95.
Vuduc, R., Demmel, J.W., Yelick, K.A., 2005. OSKI: a library of automatically tuned sparse matrix kernels. J. Phys. Conf. Series 16, 521–530.

Yzelman, A.N., Bisseling, R.H., 2009. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. SIAM J. Sci. Comput. 31 (4), 3128–3154.

Yzelman, A.N., Bisseling, R.H., 2011. Two-dimensional cache-oblivious sparse matrix–vector multiplication. Parallel Comput. 37 (12), 806–819.

Yzelman, A.N., Bisseling, R.H., 2012. A cache-oblivious sparse matrix-vector multiplication scheme based on the Hilbert curve. In: Günther, M., Bartel, A., Brunk, M., Schöps, S., Striebel, M. (Eds.), Progress in Industrial Mathematics at ECMI 2010, Mathematics in Industry. Springer, Berlin, Germany, pp. 627–633.

Yzelman, A.N., Roose, D., 2014. High-level strategies for parallel shared-memory sparse matrix-vector multiplication. IEEE Trans. Parallel Dist. Syst. 25 (1), 116–125.

Yzelman, A.N., Bisseling, R.H., Roose, D., Meerbergen, K., 2014. MulticoreBSP for C: a high-performance library for shared-memory parallel programming. Int. J. Parallel Programm. 42, 619–642.