Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors

Xing Liu
School of Computational Science and Engineering
Georgia Institute of Technology, Atlanta, Georgia
xing.liu@gatech.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology, Atlanta, Georgia
echow@cc.gatech.edu

Mikhail Smelyanskiy
Parallel Computing Lab, Intel Corporation
Santa Clara, California
mikhail.smelyanskiy@intel.com

Pradeep Dubey
Parallel Computing Lab, Intel Corporation
Santa Clara, California
pradeep.dubey@intel.com

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is an important kernel in many scientific applications and is known to be memory bandwidth limited. On modern processors with wide SIMD and large numbers of cores, we identify and address several bottlenecks which may limit performance even before memory bandwidth: (a) low SIMD efficiency due to sparsity, (b) overhead due to irregular memory accesses, and (c) load-imbalance due to non-uniform matrix structures.

We describe an efficient implementation of SpMV on the Intel® Xeon Phi™ Coprocessor, codenamed Knights Corner (KNC), that addresses the above challenges. Our implementation exploits the salient architectural features of KNC, such as large caches and hardware support for irregular memory accesses. By using a specialized data structure with careful load balancing, we attain performance on average close to 90% of KNC's achievable memory bandwidth on a diverse set of sparse matrices. Furthermore, we demonstrate that our implementation is 3.52x and 1.32x faster, respectively, than the best available implementations on dual Intel® Xeon® Processor E5-2680 and the NVIDIA Tesla K20X architecture.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; C.1.4 [**Processor Architectures**]: Parallel Architectures

Keywords

ESB format, Intel Many Integrated Core Architecture (Intel MIC), Intel Xeon Phi, Knights Corner, SpMV

1. INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is an essential kernel in many scientific and engineering applications. It performs the operation $y \leftarrow y + Ax$, where A is a sparse matrix, and x and y are dense vectors. While SpMV is considered one of the most important computational kernels, it usually performs poorly on modern architectures, achieving less than 10% of the peak performance of microprocessors [7, 26]. Achieving higher performance usually requires carefully choosing the sparse matrix storage format and fully utilizing the underlying system architecture.

Recently, Intel announced the Intel® Xeon Phi™ [1] Coprocessor, codenamed Knights Corner (KNC), which is the first commercial release of the Intel® Many Integrated Core (Intel® MIC) architecture. Unlike previous microprocessors from Intel, KNC works on a PCIe card with GDDR5 memory and offers extremely high memory bandwidth. The first model of KNC has 60 cores, featuring a new 512-bit SIMD instruction set. With a clock speed in excess of 1 GHz, KNC has over 1 Tflops double precision peak performance from a single chip.

In this paper, we consider the design of efficient SpMV kernels on KNC. We first evaluate the new architecture with a simple SpMV kernel using the widely-used Compressed Sparse Row (CSR) format. Our experiments on KNC reveal several performance bottlenecks that diverge from our prior knowledge about SpMV on other architectures. From this experience, we design a novel sparse matrix format, called ELLPACK Sparse Block (ESB), which is tuned for KNC. The new format extends the well-known ELLPACK format with (a) finite-window sorting to improve the SIMD efficiency of the SpMV kernel; (b) a bit array to encode nonzero locations to reduce the bandwidth requirement; and (c) column blocking to improve memory access locality.

With a large number of cores, one critical performance obstacle of SpMV on KNC is load imbalance. To address this problem, we present three load balancers for SpMV on KNC and compare them with other load balancing methods. Using the ESB format and the proposed load balancers, our optimized SpMV implementation achieves close to 90% of the STREAM Triad bandwidth of KNC and is 1.85x faster than the optimized implementation using CSR format. Compared to other architectures, SpMV on KNC is 3.52x faster than on dual-socket Intel® Xeon® Processor E5-2680 and is 1.32x faster than on NVIDIA Tesla K20X.

[1]Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

2. OVERVIEW OF KNIGHTS CORNER

2.1 Knights Corner Architecture

Knights Corner is a many-core coprocessor based on the Intel® MIC architecture. It consists of x86-based cores, caches, Tag Directories (TD), and GDDR5 Memory Controllers (MC), all connected by a high speed bidirectional ring. An architectural overview of KNC is given in Fig. 1.

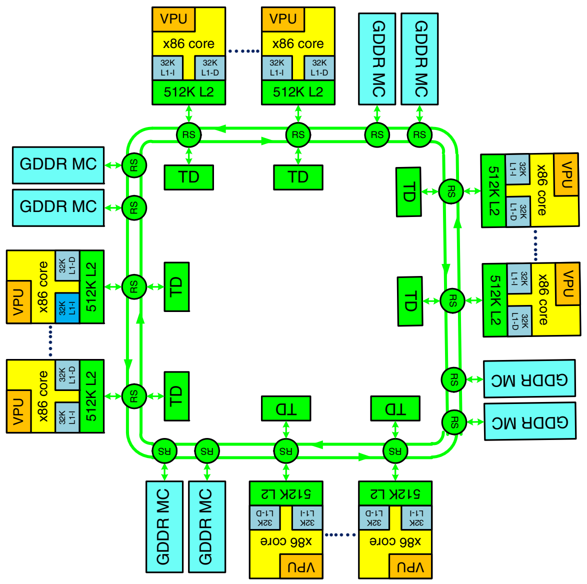

Figure 1: High-level block diagram of KNC.

Each core of KNC is composed of four components: an in-order dual-issue pipeline with 4-way simultaneous multi-threading (SMT), a 512-bit wide SIMD engine called Vector Processing Unit (VPU); 32 KB L1 data and instruction caches; and a 512 KB fully coherent L2 cache. The KNC's L2 caches are fully coherent using a set of tag directories. Each of the tag directories responds to L2 cache misses from a fixed portion of the memory space. As shown in Fig. 1, the memory controllers are symmetrically interleaved with pairs of cores around the ring. The memory addresses are uniformly distributed across the tag directories and memory controllers. This design choice provides a smooth traffic characteristic on the ring and is essential for good bus performance.

Given the large number of cores and coherent caches, L2 cache misses on KNC are expensive compared to those on other x86 processors. On a L2 cache miss, an address request is sent on the ring to the corresponding tag directory. Depending on whether or not the requested address is found in another core's cache, a forwarding request is then sent to that core or memory controllers, and the request data is subsequently forwarded on the ring. The cost of each data transfer on the ring is proportional to the distance between the source and the destination, which is, in the worse case, on the order of hundreds of cycles. Overall, the L2 cache miss latency on KNC can be an order of magnitude larger than that of multi-core CPUs.

2.2 Intel® Initial Many Core Instructions

KNC provides a completely new 512-bit SIMD instruction set called Intel® Initial Many Core Instructions (Intel® IMCI). Compared to prior vector architectures (MMX, SSE, and AVX), Intel® IMCI has larger SIMD width and introduces many new features. One important new feature is *write-mask*, which we use in our SpMV implementations. The write-mask in a SIMD instruction is a mask register coming with the destination vector. After the SIMD instruction is executed, each element in the destination vector is conditionally updated with the results of the instruction, contingent on the corresponding element position bit in the mask register.

The Intel® IMCI also supports irregular memory accesses, which are very useful for SpMV. The *gather* instruction loads sparse locations of memory into a dense vector register, while the *scatter* instruction performs the inverse. In both instructions, the load and store locations of memory are specified by a base address and a vector of signed 32-bit offsets.

2.3 Test Platform

The platform used for experimental tests in this paper is a pre-production part of KNC with codename ES B0, which is installed with Intel® MIC Platform Software Stack (MPSS) Gold 2.1. The pre-production system is equipped with 8 GB GDDR5 memory and includes a 61-core KNC coprocessor running at 1.09 GHz, which is capable of delivering 1.05 Tflops double precision peak performance.[2] In this work, we use 60 cores to test the SpMV implementations, leaving the remaining core to run the operating system and administrative software. The STREAM Triad benchmark on this system achieves a score of 163 GB/s with ECC turned on.

3. UNDERSTANDING THE PERFORMANCE OF SPMV ON KNC

We first implemented a simple SpMV kernel using CSR format on KNC. The CSR format is standard, consisting of three arrays: the nonzero values of the sparse matrix are stored in *val*; the column indices corresponding to each nonzero are stored in *colidx*; and the list of indices giving the beginning of each row are stored in *rowptr*.

3.1 Test Matrices

Table 1 lists sparse matrices used in our performance evaluation. These are all the matrices used in previous papers [26, 21, 9] that are larger than the 30 MB aggregate L2 cache of KNC (using 60 cores). A dense matrix stored in sparse format is also included. These matrices come from a wide variety of applications with different sparsity characteristics. No orderings were applied to these matrices, but our results may be better if we use orderings that promote locality when accessing the vector *x*.

Table 1: Sparse matrices used in performance evaluation.

Name	Dimensions	nnz	nnz/row
Dense8	8K×8K	64.0M	8000.00
Wind Tunnel	218K×218K	11.6M	53.39
Circuit5M	5.56M×5.56M	59.5M	10.71
Rail4284	4K×1.09M	11.3M	2633.99
Mip1	66K×66K	10.4M	155.77
In-2004	1.38M×1.38M	16.9M	12.23
Si41Ge41H72	186K×186K	15.0M	80.86
Ldoor	952K×952K	46.5M	48.86
Bone010	987K×987K	71.7M	72.63
Rucci1	1.98M×110K	7.8M	3.94
Cage15	5.16M×5.16M	99.2M	19.24
Rajat31	4.69M×4.69M	20.3M	4.33
12month1	12K×873K	22.6M	1814.19
Spal_004	10K×322K	46.1M	4524.96
Crankseg_2	64K×64K	14.1M	221.64
Torso1	116K×116K	8.5M	73.32

[2]Evaluation card only and not necessarily reflective of production card specifications.

3.2 Overview of CSR Kernel

Similar to earlier work on SpMV for multi-core CPUs [26, 7], our CSR kernel is statically load balanced by row decomposition, in which the sparse matrix is partitioned into row blocks with approximately equal numbers of nonzeros. Each thread multiplies one row block with a SIMD kernel. We also applied common optimizations to our CSR kernel, including loop unrolling and software prefetching of nonzero values and column indices.

Algorithm 1: Multiply a row block (from row startrow to endrow) in CSR format (rowptr, colidx, val).

```
1  $rowid \leftarrow$ startrow;
2  $start \leftarrow$ rowptr [$rowid$];
3  while $rowid <$ endrow do
4      $idx \leftarrow start$;
5      $end \leftarrow$ rowptr [$rowid + 1$];
6      $vec\_y \leftarrow 0$;
       /* compute for a row */
7      while $idx < end$ do
8          $rem \leftarrow end - idx$;
9          $writemask \leftarrow (rem > 8 \,?\, 0xff : (0xff \gg (8 - rem)))$;
10         $vec\_idx \leftarrow$ load (&colidx [$idx$],$writemask$);
11         $vec\_vals \leftarrow$ load (&val [$idx$], $writemask$);
12         $vec\_x \leftarrow$ gather ($vec\_idx$, &x [0], $writemask$);
13         $vec\_y \leftarrow$ fmadd ($vec\_vals$, $vec\_x$, $vec\_y$, $writemask$);
14         $idx \leftarrow idx + 8$;
15     y [$rowid$] $\leftarrow$ y [$rowid$] + reduce_add ($vec\_y$);
16     $start \leftarrow end$;
17     $rowid \leftarrow rowid + 1$;
```

Alg. 1 shows the pseudocode of the SIMD kernel, in which the inner loop (lines 7-14) processes a matrix row. In each iteration, at most 8 nonzeros are multiplied as follows. First, the elements from *colidx* and *val* are loaded into vectors (lines 10 and 11). Next, the elements from vector *x* are gathered into *vec_x* (line 12). Finally, *vec_vals* and *vec_x* are multiplied and added to *vec_y* by the fused multiply-add instruction (line 13). When fewer than 8 nonzeros are left in a row, only a portion of the slots in the SIMD instructions will be used. In this case, we only update the corresponding elements in the destination vectors. To achieve this, a write-mask (line 9) is used to mask the results of the SIMD instructions.

3.3 Performance Bounds

We use four performance bounds (measured in flops) to evaluate the performance of our CSR kernel. First, the *memory bandwidth bound performance* is the expected performance when the kernel is bounded by memory bandwidth. It is defined as

$$P_{bw} = \frac{2\,nnz}{M/B}$$

where nnz is the number of nonzeros in the matrix, B is the STREAM bandwidth of KNC, and M is the amount of memory transferred in the SpMV kernel, which is measured by Intel® VTune™.

The *compute bound performance* is measured by running a modified CSR kernel, in which all memory accesses are eliminated. Concretely, the modified kernel only uses the first cache lines of *val* and *colidx*. It also eliminates the memory accesses on vector x by setting *colidx* to be $\{0, 8, 16, 24, 32, 40, 48, 56\}$.

The *ideal balanced performance* is the achievable performance when SpMV kernels are perfectly balanced, and is computed as

$$P_{balanced} = \frac{2\,nnz}{T_{mean}}$$

where T_{mean} is the average execution time of the threads.

Finally, the *effective bandwidth bound performance* is the best performance one would expect given that (a) the kernel is bandwidth bound; (b) there are no redundant memory accesses to vectors x and y. It is defined as

$$P_{ebw} = \frac{2\,nnz}{M_{min}/B}$$

where M_{min} is the minimum memory traffic of SpMV assuming perfect reuse of vectors x and y.

Prior work has shown that SpMV is memory bandwidth bound on modern architectures [7, 26, 6]. Thus, it is expected that for all matrices the compute bound performance is larger than the memory bandwidth bound performance. The ideal balanced performance represents the achievable peak performance of a balanced SpMV kernel. For an efficient SpMV implementation, it should be close to the memory bandwidth bound performance. Additionally, the difference between the ideal balanced performance and the measured performance may quantify the degree of load imbalance.

3.4 Performance Bottlenecks

Fig. 2 shows the performance of our CSR kernel on all test matrices and the corresponding performance bounds. By comparing the measured performance of our CSR kernel to these performance bounds, we discover some unexpected performance issues.

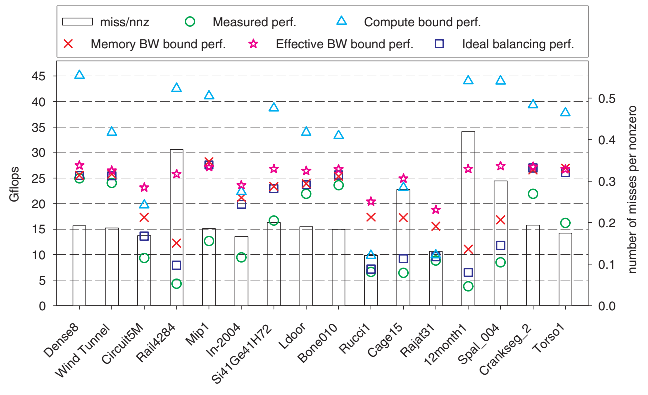

Figure 2: Performance of CSR kernel and performance bounds. The average number of L2 cache misses per nonzero is shown in vertical bar.

3.4.1 Low SIMD Efficiency

As seen in Fig. 2, for some matrices that have very short row lengths, e. g., *Rajat31* and *Rucci1*, the compute bound performance is lower than the bandwidth bound performance, suggesting that their performance is actually bounded by computation. Previous work on SpMV [7, 4] attributes the poor performance of matrices with short rows to loop overheads. For wide SIMD, we argue that this is also because of the *low SIMD efficiency* of the CSR kernel, i. e., low fraction of useful slots used in SIMD instructions.

More precisely, for a matrix with very few nonzeros per row, the CSR kernel cannot fully utilize KNC's 512-bit SIMD instructions. As a result, the CSR kernel on this matrix performs poorly, and if the SIMD efficiency is very low, the compute bound performance of the kernel may be smaller than the memory bandwidth bound performance. For example, the matrix *Rucci1* has on average 3.94 nonzeros per row. Multiplying this matrix using our CSR kernel on KNC can only use half of the SIMD slots. Thus, the SIMD efficiency is only 50%.

Low SIMD efficiency is not a unique problem for CSR. It can be shown that the compute bound performance will be lower than the memory bandwidth bound performance when the SIMD efficiency is lower than some threshold. Given a matrix format, its SpMV implementation on KNC usually includes a basic sequence of SIMD instructions similar to lines 7-14 in Alg. 1, which multiplies at most 8 nonzeros (in double precision). Assuming that the basic SIMD sequence takes C cycles (C accounts for both SIMD instructions and loop overheads) and F is the clock speed (Hz), then the compute time for the SIMD sequence is C/F. The minimum memory transfer time for the SIMD sequence is $(8s \times P \times \eta)/B$, where s is the working set size (in bytes) for one nonzero, where P is the number of cores in use, η is the SIMD efficiency, and B is the STREAM bandwidth. To avoid being bounded by computation, the compute time should be greater than the memory transfer time, thus η needs to satisfy

$$\eta > \eta_{min} = \frac{B}{8s \times P} \times C/F \tag{1}$$

This provides the minimum required SIMD efficiency for the given matrix format. We will revisit this in Section 4.

3.4.2 *Cache Miss Latency*

Four matrices, *Rail4284*, *Cage15*, *12month1* and *Spal_004*, have ideal balanced performance much lower than both the compute bound performance and the memory bandwidth bound performance. This suggests that, even if the kernel is perfectly balanced, the performance is neither bounded by computation nor bandwidth.

To determine the reason for this, Fig. 2 shows the average number of L2 cache misses (measured by Intel® VTune™) per nonzero for all test matrices. From this figure, we see that those matrices whose ideal balanced performance is lower than the compute and memory bandwidth bound performance have much more L2 cache misses than other matrices (per nonzero). We also notice that their effective bandwidth bound performance is much higher than the memory bandwidth bound performance, which means there are a large number of redundant memory accesses on vector x. This implies that the poor performance of these matrices may be because of excessive cache misses due to accessing vector x. Unlike accesses to matrix values, accesses to x are not contiguous and can be very irregular, and we do not have an effective way to perform prefetching on x to overcome cache miss latency. While KNC has hardware support for prefetching irregular memory accesses, it does have an overhead that often negates its benefits.

Cache miss latency is not a unique problem for KNC. However, it is more expensive on KNC than on other architectures. To better understand the problem, we compare the cache miss latency (memory latency) and memory bandwidth of KNC with those of CPUs and GPUs. Sandy Bridge, an example of recent multi-core CPUs, has an order of magnitude smaller latency and 5x less memory bandwidth than KNC. While the memory latency of Sandy Bridge still affects performance, the SpMV kernels on it tend to be memory bandwidth bound. High-end GPUs have comparable memory bandwidth and latency to KNC. However, memory latency is usually not a problem for GPUs as they have a large number of warps per streaming multiprocessor (vs. 4 threads per core on KNC) so the memory latency can be completely hidden.

3.4.3 *Load Imbalance*

In the CSR kernel, we statically partition the matrix into blocks with equal numbers of nonzeros. We call this partitioning scheme *static-nnz*. Although *static-nnz* is widely used in previous SpMV implementations on multi-core processors and leads to good performance, it performs poorly on KNC. As shown in Fig. 2, for at least 8 matrices the CSR kernel is highly unbalanced. This results in a significant performance degradation up to 50%.

The poor performance of *static-nnz* in these cases is mainly due to the fact that the sparsity structure of a sparse matrix often varies across different parts of the matrix. In one case, some parts of the matrix may have more rows with shorter row lengths than other parts. The threads assigned to these parts are slower than the other threads because they are burdened by more loop overheads. In another case, some parts of the matrix may have very different memory access patterns on vector x than others. For these parts of the matrix, accessing vector x may cause more cache misses. Considering that KNC has far more cores than any other x86 processor, it is more likely to have some partitions with very different row lengths or memory access patterns from others. As a result, while all partitions in *static-nnz* have the same number of nonzeros, they can have very different performance. This explains why *static-nnz* performs well on previous CPUs but is not effective on KNC.

In summary, the results using our CSR kernel on KNC reveal that, depending on the matrix, one or more of three bottlenecks can impact performance: (a) low SIMD efficiency due to wide SIMD on KNC; (b) high cache miss latency due to the coherent cache system; (c) load imbalance because of the large number of cores. We will first address the low SIMD efficiency and the cache miss latency bottlenecks in the next section by proposing a new matrix format. The subsequent section addresses load imbalance.

4. ELLPACK SPARSE BLOCK FORMAT

4.1 Motivation

From the architectural point of view, KNC bears many similarities with x86-based multi-core CPUs. On the other hand, there are also similarities between KNC and GPUs, for example, both have wide SIMD. Thus, before considering a new matrix format for KNC, it is necessary to first evaluate existing matrix formats and optimization techniques used for CPUs and GPUs.

4.1.1 *Register Blocking*

Register blocking [8] is one of the most effective optimization techniques for SpMV on CPUs, and is central to many sparse matrix formats [3, 4, 26]. In this section, we explain why this technique is not appropriate for KNC. In register blocking, adjacent nonzeros of the matrix are grouped into small dense blocks to facilitate SIMD, and to reuse portions of the vector x in registers. Given that KNC has wider SIMD and larger register files than previous x86 CPUs, it seems advantageous to use large blocks on KNC. However, sparse matrices in general cannot be reordered to give large dense blocks; zero padding is necessary, resulting in redundant computations with zeros and low SIMD efficiency.

To evaluate the use of register blocking, we measure the *average nonzero density of the blocks* (ratio of the number of nonzeros in a block to the number of nonzeros stored including padding zeros) using various block sizes and compare these with the minimum required SIMD efficiency (η_{min}) in Eq. 1. In general, the SIMD efficiency of SpMV is approximately equal to the average nonzero density. It can be shown that, even when the smallest block sizes for KNC to facilitate SIMD are used, the nonzero densities for many matrices are still smaller than η_{min}. Thus register blocking leads to low SIMD efficiency on KNC. The details are as follows.

First, we calculate η_{min} for register blocking using Eq. 1. In register blocking, all nonzeros within a block share one coordinate index, thus the working set for one nonzero is roughly equal to the number of bytes for storing one double-precision word, i. e., 8 bytes. To multiply 8 nonzeros using register blocking, an efficient implementation of the basic SIMD sequence requires approximately 6 cycles (1 cycle for loading x, 1 cycle for loading val, 1 cycle for multiplication, and 2-3 cycles for loop overheads and prefetching instructions). Additionally, we use 60 cores at 1.09 GHz, and the STREAM bandwidth on KNC achieves 163 GB/s. Putting these quantities together and using Eq 1, we obtain $\eta_{min} = 23.7\%$. We now measure the nonzero densities for the smallest block sizes that facilitate SIMD on KNC. The results are displayed in Table 2, showing that, for 4×4, 8×2 and 8×8 register blocks, 6, 7 and 8 matrices have nonzero density lower than 23.7%, respectively. We note that this is a crude study because no matrix reordering was used, but the above illustrates the point.

Table 2: Average percent nonzero density of register blocks and ELLPACK slices (slice height 8).

Matrix	Register blocking			SELLPACK
	4×4	8×2	8×4	
Dense8	100.00	100.00	100.00	100.00
Wind Tunnel	69.83	63.47	59.91	98.96
Circuit5M	29.70	21.82	18.39	60.07
Rail4284	20.09	14.00	10.75	38.18
Mip1	82.08	77.52	73.30	90.10
In-2004	37.99	37.59	28.19	66.69
Si41Ge41H72	30.30	25.38	20.45	64.88
Ldoor	59.60	53.64	45.98	89.75
Bone010	58.83	51.71	45.98	94.02
Rucci1	12.26	18.11	9.32	98.55
Cage15	17.25	12.47	9.98	97.16
Rajat31	12.22	10.98	6.67	98.73
12month1	9.82	8.86	5.66	25.39
Spal_004	23.12	14.00	13.66	91.39
Crankseg_2	52.17	48.72	40.33	89.76
Torso1	73.52	69.55	62.85	93.42

4.1.2 *ELLPACK and Sliced ELLPACK*

The ELLPACK format and its variants are perhaps the most effective formats for GPUs. ELLPACK packs the nonzeros of a sparse matrix towards the left and stores them in a $m \times L$ dense array (the column indices are stored in a companion array) where m is the row dimension of the matrix, and L is the maximum number of nonzeros in any row. When rows have fewer than L nonzeros, they are padded with zeros to fill out the dense array. The dense array is stored in column-major order, and thus operations on columns of the dense array can be vectorized with SIMD operations, making this format very suitable for architectures with wide SIMD.

The efficiency of the ELLPACK format highly depends on the distribution of nonzeros. When the number of nonzeros per row varies considerably, the performance degrades due to the overhead of the padding zeros. To address this problem, Monakov et al. [15] proposed the sliced ELLPACK (SELLPACK) format for GPUs, which partitions the sparse matrix into row slices and packs each slice separately, thus requiring less zero padding than ELLPACK.

Table 2 also shows the average nonzero density of slices in SELLPACK using slice height 8, corresponding to KNC's double precision SIMD width. For matrices with highly variable numbers of nonzeros per row, the SELLPACK format still requires excessive zero padding.

4.2 Proposed Matrix Format

SpMV in the CSR format suffers from low SIMD efficiency, particularly for matrices with short rows. Since SIMD efficiency is expected to be higher in the ELLPACK format, and since the format supports wide SIMD operations, it forms the basis of our proposed matrix format, called *ELLPACK Sparse Block* (ESB). At a high level, ESB effectively partitions the matrix coarsely by rows and columns into large sparse blocks. The row and column partitioning is a type of cache blocking, promoting locality in accessing vector x. The sparse blocks in a block column are appended and stored in a variant of the ELLPACK format.

In practice, matrices generated by an application code may have an ordering that is "local," e. g., rows corresponding to nearby grid points of a mesh are ordered together, which translates to locality when accessing the vector x. Such an ordering may also be imposed by using a bandwidth reducing ordering. The coarse row and column partitionings in ESB are intended to maintain locality, as described below.

4.2.1 *SELLPACK Storage with a Bit Array*

The original ITPACK-ELLPACK format was designed for classical vector processors which required long vector lengths for efficiency. SIMD processing on GPUs requires vector lengths equal to the number of threads in a thread block, and thus it is natural to store the matrix in slices (i. e., SELLPACK) which requires less zero padding. SIMD processing on CPUs including KNC only requires vector lengths equal to the width of SIMD registers. Again it is natural to store the matrix in slices, and the smaller slice height required on CPUs can make this storage scheme particularly effective at reducing zero padding.

To further reduce the memory bandwidth associated with the padding zeros, we can avoid storing zeros by storing instead the length (number of nonzeros) of each *row*, as done in ELLPACK-R [22] for GPUs. Here, each CUDA thread multiplies elements in a row until the end of the row is reached. For SIMD processing on CPUs, we can store the length of each *column* of the dense ELLPACK arrays, corresponding to the length of the vector operations for processing that column (this assumes that the rows are sorted by length and thus the nonzeros in each column are contiguous).

For KNC, instead of storing column lengths, it is more efficient to store a bit array, with ones indicating nonzero locations in the SELLPACK dense array with slice height 8. (Rows within a slice also do not need to be sorted by length when a bit array is used.) Each 8 bits corresponds to one column of a matrix slice and can be used expediently as a write-mask in KNC's SIMD instructions to dynamically reconstruct the SELLPACK dense array as the SpMV operation is being carried out. Required memory bandwidth is reduced with this bit array technique, but it technically does not increase SIMD efficiency because we still perform arithmetic on padding zeros. We note that earlier work has also used bit arrays for storing the sparsity pattern of blocks of sparse matrices in order to avoid explicitly storing zeros [27, 3].

4.2.2 *Finite-Window Sorting*

On GPUs, row sorting can be used to increase the nonzero density of slices for SELLPACK [5, 11]. Rows of the sparse matrix are sorted in descending order of number of nonzeros per row. As adjacent rows in the sorted matrix have similar numbers of nonzeros, storing the sorted matrix in SELLPACK format requires less zero padding than without sorting.

The adjacent rows in the sparse matrix in the original ordering often have high temporal locality on accesses to vector x. This is because the matrix either naturally has good locality or has been reordered to reduce the matrix bandwidth. While row sorting increases the nonzero density, it may destroy this locality, which results in extra cache misses. To limit this side effect of row sorting, instead of sorting the entire matrix, we partition the matrix into row blocks with block height w and sort the rows within each block individually. We call this new sorting scheme *Finite-Window Sorting* (FWS). We refer to w as the *window size*. As mentioned, this is a row partitioning that promotes locality when accessing vector x.

To evaluate FWS, we measure the average nonzero density of SELLPACK slices for various window sizes and the corresponding amount of memory transfer due to the accesses on vectors x and y (measured by Intel® VTune™) of the SpMV kernel using SELLPACK format. As an example, the result for *In-2004* is shown in Fig. 3 (results for other matrices are similar). We see that the nonzero density and the amount of memory transfer both increase as the window size increases. However, an important observation is that when the window size is smaller than some value, e. g., 1000 for *In-2004*, the amount of memory transfer appears to grow much more slowly than the nonzero density. This is a positive result since it suggests that we may use FWS to significantly increase the nonzero density while only slightly increasing the amount of memory transfer.

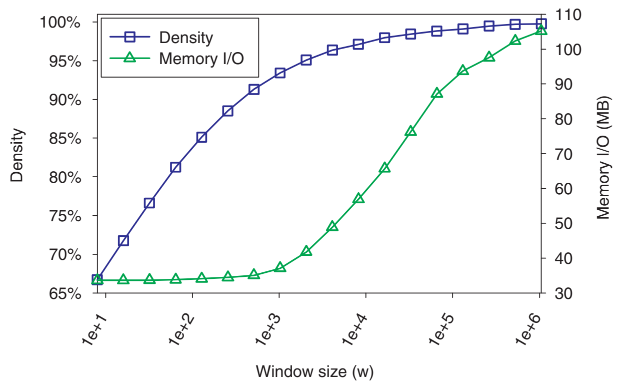

Figure 3: Nonzero density and memory I/O as a function of window size (w) for *In-2004*.

To address the low SIMD efficiency problem of SELLPACK, we use FWS to increase the average nonzero density of slices to be just larger than η_{min}. Since η_{min} is relatively small, a small w is sufficient for most matrices, thus the increase in memory transfer is very slight. We use the same w for the entire matrix.

Fig. 4 illustrates the ESB format for a $m \times n$ matrix. The ESB format consists of six arrays: *val*, the nonzero values of the matrix; *colidx*, the column indices for the nonzeros; *vbit*, an array of bits marking the position of each nonzero in SELLPACK; *yorder*, the row ordering after FWS (c permutation vectors of length m); *sliceptr*, a list of $m/8 \times c$ values indexing where each slice starts in *val*; *vbitptr*, a list of c values indexing where each column block starts in *vbit*.

4.3 SpMV Kernel with ESB Format

The multiplication of a column block of ESB matrix is illustrated in Alg. 2. Due to the use of the bit array, every SIMD instruction in the inner loop, which processes one slice, has a write-mask. At the end of each inner iteration, the *popcnt* instruction (line 17) is used to count the number of 1's in *vbit*, which is equal to the number of nonzeros processed in this iteration. Since the rows of the matrix have been permuted, we use gather and scatter instructions to load and store y, using the offsets stored in *yorder*.

In practice, we also parallelize the SpMV operation across column blocks. Here, we use a temporary copy of y for each block and use a reduce operation across these temporary copies at the end of the computation.

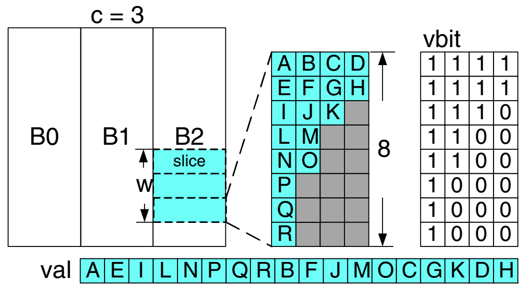

Figure 4: ELLPACK sparse block format. The sparse matrix is partitioned into c column blocks. Each column block is sorted by FWS with window size w and stored in SELLPACK format (slice height = 8) with a bit array.

The reconstruction of the dense arrays are implemented using load unpack instructions (lines 13 and 14), which are able to load contiguous elements from memory and write them sparsely into SIMD registers. The write-mask in the load unpack instruction is used to mark which positions of the SIMD register to write the loaded elements.

Algorithm 2: Multiply the *i th* column block of matrix in ESB format.

```
1  $yr \leftarrow i \times m$;
2  $startslice \leftarrow i \times m/8$;
3  $endslice \leftarrow (i+1)m/8$;
4  $sliceidx \leftarrow startslice$;
5  $idx \leftarrow$ sliceptr [$sliceidx$];
6  while $sliceidx < endslice$ do
7      $k \leftarrow$ vbitptr [$i$];
8      $end \leftarrow$ sliceptr [$sliceidx + 1$];
       /* compute for a slice */
9      $vec\_offsets \leftarrow$ load (&yorder [$yr$]);
10     $vec\_y \leftarrow$ gather ($vec\_offsets$, &y [0]);
11     while $idx < end$ do
12         $writemask \leftarrow$ vbit [$k$];
13         $vec\_idx \leftarrow$ loadunpack (&colidx [$idx$], $writemask$);
14         $vec\_vals \leftarrow$ loadunpack (&val [$idx$], $writemask$);
15         $vec\_x \leftarrow$ gather ($vec\_idx$, &x [0], $writemask$);
16         $vec\_y \leftarrow$ fmadd ($vec\_vals$, $vec\_x$, $vec\_y$, $writemask$);
17         $idx \leftarrow idx +$ popcnt (vbit [$k$]);
18         $k \leftarrow k + 1$;
19     scatter ($vec\_y$, $vec\_offsets$, &y [0]);
20     $yr \leftarrow yr + 8$;
21     $sliceidx \leftarrow sliceidx + 1$;
```

Given the ESB kernel shown in Alg. 2, we can calculate η_{min} for the ESB format. From the ESB kernel assembly code, the basic SIMD sequence (lines 11-18) takes approximately 26 cycles. In ESB format, the working set for one nonzero includes one value and one column index, i. e., 12 bytes. Using Eq. 1, we have that the SIMD efficiency of any SpMV kernel using the ESB format needs to be larger than 68.3%.

4.4 Selecting *c* and *w*

To select the number of block columns c and the window height w, several values are tested for each matrix. Although c and w are not independent, we find that they are only weakly related and that it is better to select c before selecting w.

The choice of number of block columns c is dependent on the structure of the sparse matrix. Nishtala et al. [16] describes situations where cache blocking is beneficial, including the presence of very long rows. We found column blocking to be beneficial for three of our test matrices: *Rail4284*, *12month1* and *Spal_004*.

For matrices that favor cache blocking, the execution time of SpMV can be modeled as $T(c) = (12nnz + 16mc + M_x(c))/B$, where the first two terms in the numerator represent the memory traffic due to accessing the matrix and vector y respectively, and $M_x(c)$ is the memory traffic due to accessing vector x. Since M_x decreases as c increases and m is relatively small, this model suggests that the execution time of the SpMV kernel might first decrease as c decreases until a certain value, near where there is no more temporal locality of vector x to exploit. After that point, the execution time will increase since the memory traffic due to accessing vector y increases as c increases.

Fig. 5 shows the performance of the ESB kernel for the three matrices that favor cache blocking as c varies between 1 and 1024, which confirms the implication of our performance model. The figure also shows that the performance decreases slowly when c is larger than the optimal value. We thus use the heuristic that we can safely choose c as a power of 2.

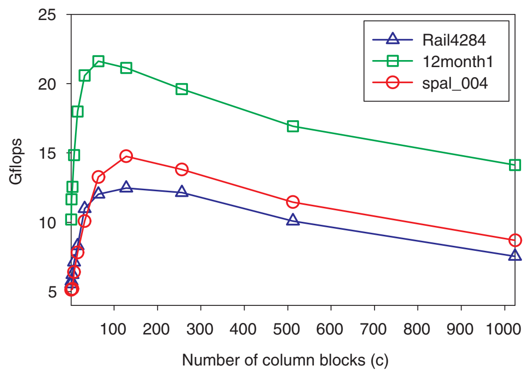

Figure 5: Performance of ESB kernel for *Rail4284 12month1*, and *Spal_004* as a function of the number of column blocks (c).

We select the window size w by gradually increasing it until the nonzero density of slices exceeds η_{min}. As discussed in Section 4.3, η_{min} of ESB format is 68.3%, however, we use 75% to be more conservative. The cost of search can be reduced by a matrix sampling scheme, proposed in [8] for choosing register block sizes. In the matrix sampling scheme, 1% of the slices are selected from the SELLPACK matrix to form a new matrix, and the new matrix is sorted to estimate the nonzero densities for various w.

5. LOAD BALANCERS FOR KNC

SpMV is parallelized across 240 threads on KNC, or 4 threads per core. To assign work to each thread, we consider three partitioning/load balancing schemes. The starting point for these schemes is a matrix in ESB format, where parameters such as c, w, and matrix ordering are considered fixed. We note that these balancers are general and may be applied to other KNC workloads.

5.1 Static Partitioning of Cache Misses

The performance of SpMV kernels highly depends on the amount of memory transfer and the number of cache misses. This leads to our first load balancer, called *static-miss*. Here, the sparse matrix is statically partitioned into row blocks that have equal numbers of L2 cache misses. This is accomplished approximately using a simple cache simulator which models key characteristics of the L2 cache of a single core of KNC: total capacity, line size and associativity.

The main drawback of *static-miss* is the high cost of cache simulation for each matrix of interest. In our implementation, the cost is greater than that of a single matrix-vector multiplication, which means that the application of *static-miss* is limited to cases where the SpMV kernel is required to execute a large number of times with the same matrix. Like other static partitioning methods, *static-miss* also does not capture dynamic runtime factors that affect performance, e. g., the number of cache misses is sometimes highly dependent on thread execution order.

5.2 Hybrid Dynamic Scheduler

A more sophisticated load balancing method for SpMV on KNC is to use dynamic schedulers, which are categorized into two types: work-sharing schedulers and work-stealing schedulers. However, both types have performance issues on KNC. For work sharing schedulers, the performance bottleneck is the contention for the public task queue. Due to the large number of cores, data contention on KNC is more expensive than on multi-core CPUs, especially when data is shared by many threads. On the other hand, work stealing schedulers on KNC suffer from the high cost of stealing tasks from remote threads. The stealing cost is proportional to the distance between the thief thread and the victim thread.

To address the performance issues of both work-sharing and work-stealing schedulers, we design a hybrid work-sharing/work-stealing scheduler, inspired by [18]. In the hybrid dynamic scheduler, the sparse matrix is first partitioned into P tasks with equal numbers of nonzeros. The tasks are then evenly distributed to N task queues, each of which is shared by a group of threads (each thread is assigned to a single queue). Within each thread group, the work-sharing scheduler is used to distribute tasks. Since each task queue is shared by a limited number of threads, the contention overhead is lowered. When a task queue is empty, the threads on the queue steal tasks from other queues. With this design, the number of stealing operations and the overhead of stealing are reduced.

The two parameters P and N need to be carefully selected to achieve high performance. P controls the maximum parallelism and also affects the temporal locality of memory accesses. With smaller tasks, the SpMV kernel will lose the temporal locality of accessing vector x. For our test matrices, P is chosen in the range of 1000-2000, based on the size of the sparse matrix. N is a tradeoff between the costs of work sharing and work stealing and is chosen to be 20 in our SpMV implementation, i. e., 12 threads (3 cores) share one task queue.

5.3 Adaptive Load Balancer

In many scientific applications, such as iterative linear equation solvers, the SpMV kernel is executed multiple times for the same matrix or matrix sparsity pattern. For these applications, it is possible to use performance data from earlier calls to SpMV to repartition the matrix to achieve better load balance.

This idea, which we call adaptive load balancing, was first proposed for SpMV by Lee and Eigenmann [12]. In their load balancer, the execution time of each thread is measured after each execution of SpMV. The normalized cost of each row of the sparse matrix is then approximated as the execution time of the thread to which the row is assigned, divided by the total number of rows assigned to that thread. For the next execution of SpMV, the sparse matrix is re-partitioned by evenly dividing costs of rows.

Although Lee and Eigenmann's method was originally designed for distributed memory systems, we adapted it to KNC with minor changes. To reduce the cost of re-partitioning, we used the 1D partitioning algorithm of Pinar and Aykanat [19]. Experiments show that the adaptive tuning converges in fewer than 10 executions.

6. EXPERIMENTAL RESULTS

6.1 Load Balancing Results

We first test the load balancing techniques discussed in Section 5 for parallelizing SpMV for matrices in ESB format. We compare the following load balancers:

- *static-nnz*: statically partition the matrix into blocks with approximately equal numbers of nonzeros;
- *static-miss*: statically partition the matrix into blocks with approximately equal numbers of cache misses;
- *work-sharing*: working-sharing dynamic scheduler;
- *work-stealing*: working-stealing dynamic scheduler;
- *hybrid-scheduler*: hybrid dynamic scheduler;
- *adaptive-balancer*: adaptive load balancer after 5 tuning executions.

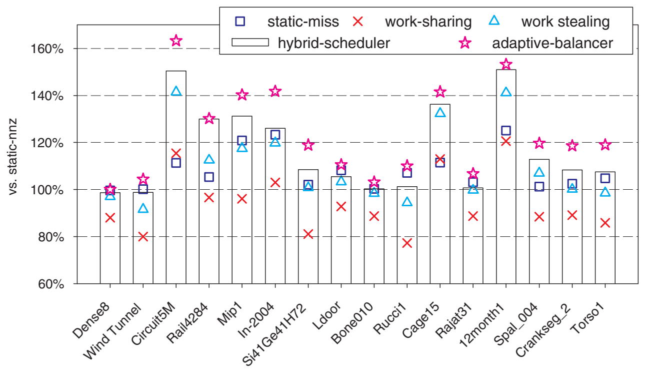

Figure 6: Normalized performance of SpMV using various load balancers.

Fig. 6 shows the results, which are normalized against those of *static-nnz*. As evident in the figure, *adaptive-balancer* gives the best SpMV performance, achieving on average 1.24x improvement over *static-nnz*. The *hybrid-scheduler* has best SpMV performance among all the dynamic schedulers, but is 5% worse than *adaptive-balancer*, likely due to scheduling overheads. Results for *hybrid-scheduler* are generally better than those for *work-stealing* and *work-sharing*. The *work-sharing* scheduler is worst for most matrices and only outperforms *static-nnz* and *static-miss* on the largest matrices, e. g., *Circuit5M* and *Cage15*, for which data contention overheads can be amortized.

The static partitioning methods have an advantage over dynamic schedulers when the matrices have regular sparsity patterns, such as even distributions of the number of nonzeros per row, as in *Wind Tunnel* and *Rajat31*. We found *static-miss* to give better performance than *static-nnz* for all the matrices, but its results are not uniformly good, likely because of the poor accuracy of our simple cache simulator for irregularly structured matrices.

6.2 ESB Results

SpMV performance results using ESB format are presented in Fig. 7. In these results, optimal values of w and c were selected using the method described in Section 4.4. The timings do not include time for selecting these parameters or for FWS. The hybrid dynamic scheduler was used for load balancing.

To quantify the effect of column blocking (CB), and the use of bit arrays and FWS, we also test SpMV kernels that incrementally implement these techniques. In the figure, green bars show SpMV performance using the SELLPACK format, which is our baseline. Yellow bars show the improvement by using column blocking only (applied to three matrices). Orange bars show the additional improvement by using bit arrays. Red bars show the performance using the full set of ESB optimizations.

We also use pink squares to show results using complete row sorting (no windows used for sorting). Finally, blue triangles are used to show the performance of the CSR kernel.

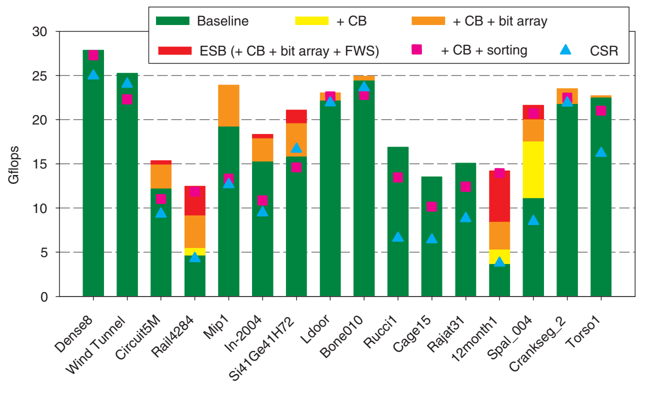

Figure 7: Performance of SpMV implementations using ESB format, showing increasing levels of optimizations of column blocking, bit array and finite-window sorting.

Although column blocking gives an improvement for the three matrices with very long rows, the improvement in two cases is very modest. A disadvantage of column blocking is that it can decrease the average nonzero density of the blocks and thus decrease SIMD efficiency.

The bit array technique improves performance in most cases, however, a small performance degradation ($< 3\%$) is also observed for some matrices, including *Dense8*, *Rucci1* and *Cage15*. This is expected for matrices with nonzero densities that are already large, since the bit array introduces additional overhead while there is no room to reduce bandwidth.

The figure also shows the significance of using FWS, which helps 6 matrices that have nonzero density lower than η_{min}. Due to its destruction of locality, complete sorting leads to poor results on KNC, with results that are poorer than the baseline for many matrices. This justifies the use of FWS.

The final ESB implementation outperforms the CSR implementation, as to be expected. On average, ESB is 1.85x faster than CSR. The greatest speedups come from the matrices with low SIMD efficiency, cache miss latency problems, and/or load imbalance.

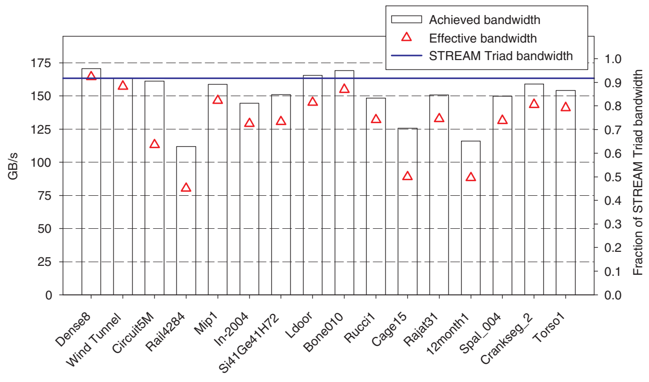

Figure 8: Achieved and effective bandwidth of SpMV implementation using ESB format.

Fig. 8 shows the achieved and the effective bandwidth of our ESB kernel. The achieved bandwidth was measured by Intel® VTune™. The effective bandwidth, which is used in prior work [26, 5], accounts only for the "effective" memory traffic of the kernel. For a sparse matrix with m rows, n columns and nnz nonzeros, the optimistic flop:byte ratio of SpMV in ESB format is $\lambda = 2\,nnz/(12\,nnz + 16\,m + 8\,n)$, which excludes redundant memory accesses to vectors x and y. Assuming SpMV on the matrix achieves P flops, the effective bandwidth is P/λ.

By comparing the achieved and the effective bandwidth with the STREAM bandwidth, we see how far the performance of our kernel is from the upper performance bound. On average, the achieved bandwidth and effective bandwidth are within 10% and 20% of the STREAM Triad bandwidth, respectively, indicating that the implementation is very efficient.

6.3 Performance Comparison

We now compare the performance of SpMV on KNC with SpMV performance on the following architectures:

- *NVIDIA K20X*: NVIDIA Tesla K20X (Kepler GPU), 6 GB GDDR5 memory, 1.31 Tflops peak double precision performance, 181 GB/s (ECC on) STREAM Triad bandwidth;
- *Dual SNB-EP*: 2.7 GHz dual-socket Intel® Xeon® Processor E5-2680 (Sandy Brige EP), 20 MB L2 cache, 32 GB DDR memory, 346 Gflops peak double precision performance, 76 GB/s STREAM Triad bandwidth.

For K20X, we used two SpMV codes: cuSPARSE v5.0[3] and CUSP v0.3 [2], which use a variety of matrix formats, CSR, COO, DIA, BCSR, ELLPACK, and ELL/HYB [1]. For each of the test matrices, we ran experiments with both codes using all the formats. For brevity, we only report the best of these results. For the dual SNB-EP system, we tested three SpMV codes: the CSR format implementation from Intel® MKL v11.0, the auto-tuning implementation from Williams et al. [26] and the Compressed Sparse Block (CSB) implementation (2011 release) from Buluç et al. [3]. Again, only the best performance of the three implementations is reported. For comparison purposes, ECC was turned on in all the platforms.

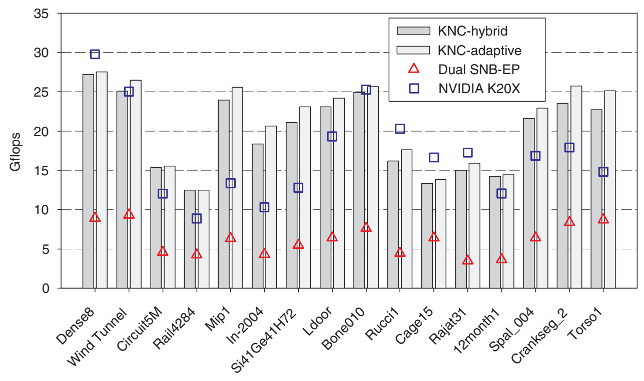

Figure 9: Performance comparison of SpMV implementations on KNC, SNB-EP and GPUs.

Fig. 9 shows the comparative results. KNC has a large advantage over dual SNB-EP; for most matrices, *KNC-adaptive* and *KNC-hybrid* are 3.52x and 3.36x faster, respectively, than SNB-EP, which is expected since KNC has significantly higher memory bandwidth. Compared to the K20X, although KNC has a lower STREAM bandwidth, *KNC-adaptive* is on average 1.32x faster than the best SpMV implementation on K20X. We believe that this mainly due to KNC having much larger caches than K20X, so accesses to vectors x and y are more likely to be directed to caches rather than memory. For the matrices with "regular" sparsity patterns, e. g., banded matrices, which are easily handled by small caches, KNC has little performance advantage over or is slower than K20X. For matrices that have complex sparsity patterns, KNC's advantage over K20X can be as high as 2x.

[3]https://developer.nvidia.com/cusparse

7. RELATED WORK

SpMV optimization has been extensively studied over decades on various architectures. Relevant for us is optimizations for CPUs and GPUs. For a comprehensive review, we refer to several survey papers [23, 8, 26, 7].

Blocking is widely used for optimizing SpMV on CPUs. Depending on the motivation, block methods can be divided into two major categories. In the first category, blocking improves the spatial and temporal locality of the SpMV kernel by exploiting data reuse at various levels in the memory hierarchy, including register [13, 8], cache [8, 16] and TLB [16]. In the second category, block structures are discovered in order to eliminate integer index overhead, thus reducing the bandwidth requirements [24, 20]. Besides blocking, other techniques have also been proposed to reduce the bandwidth requirements of SpMV. These techniques broadly include matrix reordering [17], value and index compression [25, 10], and exploiting symmetry [3].

Due to the increasing popularity of GPUs, in recent years numerous matrix formats and optimization techniques have been proposed to improve the performance of SpMV on GPUs. Among the matrix formats, ELLPACK and its variants have been shown to be most successful. The first ELLPACK-based format for GPUs was proposed by Bell and Garland [1], which mixed ELLPACK and COO formats. Monakov et al. [15] invented the Sliced ELLPACK format, in which slices of the matrix are packed in ELLPACK format separately, thus reducing zero padding. Vázquez et al. [22] used another approach to address zero-padding. Here, in ELLPACK-R, all the padding zeros are removed, and a separate array is used to store the length of each row. Kreutzer et al. [11] proposed the pJDS matrix format, which extended ELLPACK-R by row sorting. There are also blocked SpMV implementations on GPUs. Jee et al. [5] proposed the BELLPACK matrix format which partitions the matrix into small dense blocks and organizes the blocks in ELLPACK format. Monakov et al. [14] proposed a format also using small dense blocks, but augments it with ELLPACK format for nonzeros outside this structure.

8. CONCLUSIONS

This paper presented an efficient implementation of SpMV for the Intel® Xeon Phi™ Coprocessor. The implementation uses a specialized ELLPACK-based format that promotes high SIMD efficiency and data access locality. Along with careful load balancing, the implementation achieves close to optimal bandwidth utilization. The implementation also significantly outperforms an optimized implementation using the CSR format.

There has been much recent work on SpMV for GPUs, and high performance has been attained by exploiting the high memory bandwidth of GPUs. However, GPUs are not designed for irregularly structured computations, such as operations on sparse matrices with nonuniform numbers of nonzeros per row. For our test set, we find our SpMV implementation on KNC to perform better on average than the best implementations currently available on high-end GPUs. One general explanation is that KNC has much larger caches than GPUs, which helps reduce the cost of irregular accesses on vector *x*. Also, additional hardware support on KNC, such as load unpack, enables more delicate optimization techniques, e. g., the bit array technique used in this paper, than would be possible on GPUs without such support. Indeed, we expect that the many compression techniques for SpMV proposed in the literature (and independent of the ideas of this paper) can further reduce required bandwidth and improve the performance of our SpMV implementation, but these may be difficult to implement on GPUs.

The general performance issues raised in this paper for SpMV also apply to other workloads. Due to its large SIMD width, achieving high performance on KNC will require carefully designing algorithms to fully utilize SIMD operations, even for applications that are memory bandwidth bound. Careful attention to data locality and memory access patterns can help minimize the performance impact of high cache miss latencies. Our experience with SpMV also demonstrates the importance of a well-designed load balancing method.

9. ACKNOWLEDGMENTS

The authors would like to thank Aydin Buluç, Lenny Oliker and Samuel Williams for helpful discussions and for providing us with their SpMV implementations for performance comparisons.

10. REFERENCES

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. ACM/IEEE Conf. Supercomputing*, SC '09, pp. 18. ACM, 2009.

[2] N. Bell and M. Garland. CUSP: Generic parallel algorithms for sparse matrix and graph computations, 2012. V0.3.0.

[3] A. Buluc, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. 2011 IEEE Intl Parallel & Distributed Processing Symposium*, IPDPS 2011, pp. 721–733, Washington, DC, USA, 2011. IEEE.

[4] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. *Intl J. High Perf. Comput. Appl.*, 21:467–484, 2007.

[5] J. Choi, A. Singh, and R. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Notices*, volume 45, pp. 115–126. ACM, 2010.

[6] J. Davis and E. Chung. SpMV: A memory-bound application on the GPU stuck between a rock and a hard place. *Microsoft Technical Report*, 2012.

[7] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *J. Supercomput.*, 50:36–77, 2009.

[8] E. Im, K. Yelick, and R. Vuduc. SPARSITY: Optimization framework for sparse matrix kernels. *Intl J. High Perf. Comput. Appl.*, 18:135–158, 2004.

[9] V. Karakasis, G. Goumas, and N. Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *Proc. 2009 Intl Conf. Comput. Sci. and Eng.*, CSE '09, pp. 247–256. IEEE, 2009.

[10] K. Kourtis, G. Goumas, and N. Koziris. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Trans. Architecture and Code Optimization*, 7:16, 2010.

[11] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Proc. 2012 IEEE Intl Parallel and Distributed Processing Symposium Workshops*, IPDPS 2012, pp. 1696–1702, Washington, DC, USA, 2012. IEEE.

[12] S. Lee and R. Eigenmann. Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems. In *Proc. 22nd Intl Conf. Supercomputing*, ICS '08, pp. 195–204. ACM, 2008.

[13] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Intl J. of High Perf. Comput. Appl.*, 18:225–236, 2004.

[14] A. Monakov and A. Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *Proc. 9th Intl Workshop Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pp. 289–297, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proc. 5th Intl Conf. High Perf. Embedded Architectures and Compilers*, HiPEAC'10, pp. 111–125, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput.*, 18:297–311, 2007.

[17] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44:373–393, 2002.

[18] S. Olivier, A. Porterfield, K. Wheeler, and J. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proc. 1st Intl. Workshop Runtime and Operating Systems for Supercomputers*, ROSS '11, pp. 49–56. ACM, 2011.

[19] A. Pinar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *J. Parallel Distrib. Comput.*, 64:974–996, 2004.

[20] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. ACM/IEEE Conf. Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[21] B. Su and K. Keutzer. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proc. 26th Intl Conf. Supercomputing*, ICS '12, pp. 353–364. ACM, 2012.

[22] F. Vázquez, J. Fernández, and E. Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurr. Comput.: Pract. Exper.*, 23:815–826, 2011.

[23] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, Univ. of California, Berkeley, 2003.

[24] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proc. High-Perf. Comput. and Commun. Conf.*, HPCC'05, pp. 807–816, Sorrento, Italy, 2005.

[25] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proc. 20th Intl Conf. Supercomputing*, ICS '06, pp. 307–316. ACM, 2006.

[26] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Conf. Supercomputing*, SC '07, pp. 38:1–38:12, New York, NY, USA, 2007. ACM.

[27] S. W. Williams. *Auto-tuning performance on multicore computers*. PhD thesis, Univ. of California, Berkeley, 2008.