Vectorized Parallel Sparse Matrix-Vector Multiplication in PETSc Using AVX-512

Hong Zhang
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL, USA
hongzhang@anl.gov

Richard T. Mills
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL, USA
rtmills@anl.gov

Karl Rupp
Institute for Microelectronics
TU Wien
Wien, Austria
me@karlrupp.net

Barry F. Smith
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL, USA
bsmith@mcs.anl.gov

ABSTRACT

Emerging many-core CPU architectures with high degrees of single-instruction, multiple data (SIMD) parallelism promise to enable increasingly ambitious simulations based on partial differential equations (PDEs) via extreme-scale computing. However, such architectures present several challenges to their efficient use. Here, we explore the efficient implementation of sparse matrix-vector (SpMV) multiplications—a critical kernel for the workhorse iterative linear solvers used in most PDE-based simulations—on recent CPU architectures from Intel as well as the second-generation Knights Landing Intel Xeon Phi, which features many CPU cores, wide SIMD lanes, and on-package high-bandwidth memory. Traditional SpMV algorithms use compressed sparse row storage format, which is a hindrance to exploiting wide SIMD lanes. We study alternative matrix formats and present an efficient optimized SpMV kernel, based on a sliced ELLPACK representation, which we have implemented in the PETSc library. In addition, we demonstrate the benefit of using this representation to accelerate preconditioned iterative solvers in realistic PDE-based simulations in parallel.

KEYWORDS

parallel SpMV, PETSc, vectorization, many-core, Xeon Phi

ACM Reference Format:
Hong Zhang, Richard T. Mills, Karl Rupp, and Barry F. Smith. 2018. Vectorized Parallel Sparse Matrix-Vector Multiplication in PETSc Using AVX-512 . In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225100

1 INTRODUCTION

The numerical solution of partial differential equations (PDEs) often requires the solution of large sparse linear systems. The standard algorithms to solve these systems are preconditioned Krylov subspace solvers, where the computational cost is often dominated by sparse matrix-vector (SpMV) multiplications in the application of the linear operator and within the preconditioner [6]. The performance of these operations can be strongly affected by the sparse matrix storage format, the optimal choice of which depends on the underlying hardware and the matrix structure.

Many research efforts in high-performance computing (HPC) aim to design a single format or framework to deliver good performance for all applications, focusing on performance engineering for a benchmark using a collection of matrices arising from highly diverse fields. However, a single format is unlikely to suffice for the performance and algorithmic needs of versatile scientific computing libraries such as the Portable Extensible Toolkit for Scientific computation (PETSc) [1]. PETSc is used for diverse scientific computing tasks but is most often employed in applications based on partial differential equations (PDEs), where matrices typically result from spatial discretization using finite-element, finite-volume, or finite-difference methods that often have characteristic diagonal or off-diagonal sparse structure. Therefore, in order to exploit the special characteristics of these matrices, a variety of matrix formats such as AIJ (compressed sparse row – CSR), BAIJ (block CSR – BCSR), SBAIJ (symmetric BCSR), AIJPERM, AIJCUSPARSE, and AIJVIENNACL have been developed in PETSc in the past two decades. Although CSR is the most widely used among PETSc applications because of its generality and its suitability for standard CPUs, the others either take advantage of the block or symmetric structure (BAIJ and SBAIJ) or are designed for specific platforms such as vector computers (AIJPERM) and graphics processing units (GPUs) (AIJCUSPARSE, AIJVIENNACL).

Sparse linear algebra libraries often support operations similar to level 1 and level 2 BLAS. Because of the low arithmetic intensity of these sparse kernels, tuning or optimizing their performance on many-core architectures [24], such as GPUs, is generally focused on achieving optimal memory bandwidth, rather than the complicated register- and cache-centric optimizations required for compute-intensive dense kernels. The Intel Xeon Phi architecture, especially

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
ICPP 2018, August 13–16, 2018, Eugene, OR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225100

the current generation (Knights Landing, or KNL), offers high memory bandwidth and a large amount of on-chip parallelism, presenting a good opportunity to improve the performance for large-scale memory-bound applications. Thousands of KNL nodes are used in production on current large-scale supercomputers (Theta at Argonne National Laboratory and Cori at NERSC).

Intel Xeon Phi processors feature substantially more cores than standard Xeon processors, configurable on-chip interconnect, wide vector units, and on-package MCDRAM (a high-bandwidth memory) [2]. The Xeon Phi uses the same programming models as its multicore x86 predecessors, making application migration and porting straightforward. However, achieving high performance usually requires making efficient use of wide vector units. Generating vector code can be accomplished with autovectorization by the compiler for simple loops, but more complex loop structures may require manual efforts such as refactoring the code and using Intel intrinsic functions. In this work we discuss the performance optimization of the parallel SpMV kernel for PETSc native matrix formats and introduce the design and implementation of a sliced ELLPACK format and an efficient SpMV kernel based on it. We investigate the impact of these sparse matrix formats and key hardware features using representative applications that are solved with sophisticated linear solvers and preconditioners.

2 BACKGROUND

PETSc features a hierarchy of linear, nonlinear, and timestepping objects that can be combined in a composable manner, which enabling users to build their applications at a different levels of abstraction (Figure 1). Near the bottom of the hierarchy are parallel data structures based on MPI. In this section, we first introduce the existing matrix formats in PETSc and the most important linear algebra kernel based on them. We then briefly discuss the related work for a more SIMD-friendly format named ELLPACK. We also describe the specifications of the KNL architecture and some observations from benchmark tests.

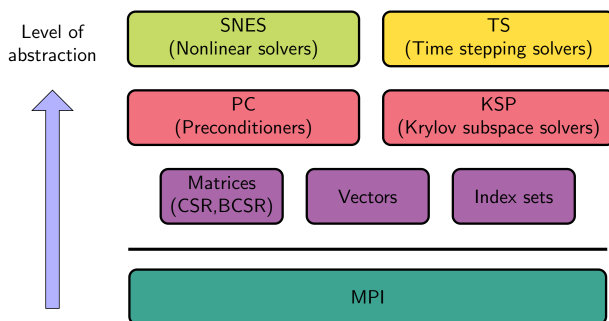

Figure 1: Hierarchical structure of the PETSc library.

2.1 Parallel sparse matrices in PETSc

In PETSc a parallel sparse matrix is generally distributed across different processors by row, with each process storing the corresponding portion as two matrices, one containing the square "diagonal block" and the other containing everything else (combined virtually into an "off-diagonal block"). Figure 2 shows a sketch of the matrix storage layout. This partitioning strategy enables elegant reuse of the kernels written for the sequential versions so as to improve developer productivity and software maintainability.

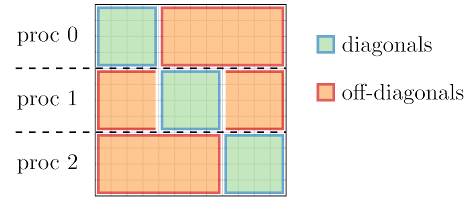

Figure 2: Storage layout for a PETSc sparse matrix.

2.2 Sparse matrix-vector multiplication

Sparse matrix-vector multiplications are the dominant components of both PETSc Krylov space iterative solvers and many preconditioners used together with the solvers, such as multigrid and polynomial preconditioners.

In parallel, each process owns a consecutive row block of the matrix and a portion of the input vector that corresponds to these rows. The parallel SpMV kernel reuses the sequential SpMV kernel for the diagonal block and overlaps the communication of nonlocal vector entries with computation. In particular, it takes the following steps:

(1) Send nonblocking request for the nonlocal data of the vector on other processors;
(2) Multiply the diagonal block with the local portion of the vector using the sequential SpMV kernel;
(3) Wait for the data transfer to complete;
(4) Multiply the off-diagonal block using the sequential SpMV kernel.

The off-diagonal matrix block is typically sparse and contains only a few nonzero rows and columns. In this case, not a full CSR-representation of the off-diagonal block is used, but only the nonzero rows are stored ("compressed CSR"). Consequently, the SpMV kernel for the diagonal part of the local matrix usually dominates the execution time.

2.3 CSR format

A CSR representation comprises three arrays: a dense array storing nonzero matrix elements rowwise (val), an array indicating the position of the first nonzero at each row (rowptr), and an array of column indices (colidx), as illustrated in Figure 3.

CSR-based SpMV suffers from the following drawbacks:

(1) **Loop remainder** Vectorized implementation of the CSR SpMV kernel will have to deal with a remainder whenever the number of nonzeros in a row is not a multiple of the SIMD vector length (8 for AVX512 in double precision). A performance penalty will be incurred regardless of whether the remainder loop is vectorized or not. The penalty becomes more severe when each row has just a few nonzeros. In general, efficient execution of the CSR SpMV requires that the number of nonzeros per row be divisible by the vector

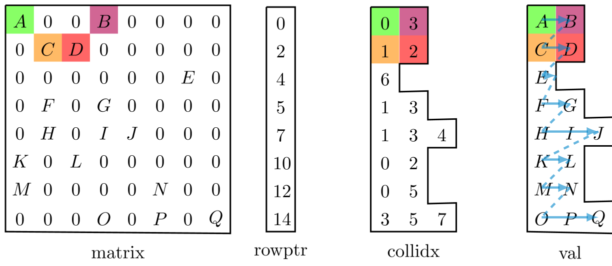

Figure 3: Representation for the compressed sparse row format.

length or large enough to mitigate the penalty due to the remainder.

(2) **Data locality** The CSR SpMV kernel allows for consecutive data access to the matrix and the output vector: the inner loop iterates over elements in each row, which are stored consecutively in memory; the outer loop runs over the matrix row by row, writing to the output vector contiguously. However, the accessing pattern for the input vector in the inner loop depends on the column indices of the nonzero entries in the matrix [21]. If the nonzeros in a row are sparsely distributed, the vectorized load operation will produce nonconsecutive accesses to the input vector. This situation can occur even with common matrices with fairly regular sparsity structure, such as banded matrices resulting from finite difference or finite element discretization.

2.4 CSR with permutation

An algorithm that performs SpMV in CSR format without data rearrangement was presented in [7]. It was implemented in PETSc as a variant of the CSR format. The key idea is to keep data in place and access it indirectly by using an extra permutation vector. The SpMV kernel based on CSR with permutation is vectorized across the row index just as in ELLPACK, resulting in irregular access to the value array and the column index array. But when there are many rows with the same number of nonzeros and nonunit stride access is fast (which was the case for Cray X1-series vector machines), the vectorization can still be effective.

2.5 ELLPACK formats

The ELLPACK format and its variants (such as ELLPACK-R [23], pJDS) were originally developed for early vector processor machines and have a history of successful use on GPUs [3–5, 13, 17, 25, 26]. The basic idea of ELLPACK is to shift the nonzero entries of a sparse matrix to the left and store them column by column in a $m \times L$ dense array, where L is the maximum number of zeros in any row. Padded zeros are used in rows with fewer than L nonzeros. The long column length of the dense array allows for exploitation of efficient vectorization on architectures with wide SIMD (single instruction, multiple data) or SIMT (single instruction, multiple threads) such as GPUs. ELLPACK-R [23] adds an additional array that stores the length of every row in order to avoid conditional branches and reduce unneeded computation when computing SpMV on GPUs. Bell and Garland [3] proposed a hybrid storage format that uses ELLPACK for most of the matrix and coordinate format for any remaining nonzero elements. This approach can reduce the overhead in pure ELLPACK when the number of nonzeros per row is uneven. Sliced ELLPACK, proposed by Monakov et al. [17], partitions the matrix into slices of C adjacent rows where C is a tunable parameter that allows a trade-off between the storage penalty and vectorization efficiency. Each slice is essentially a submatrix in ELLPACK format, and the maximum number of zeros L for each slice can be different, making the matrix substantially more compact than ELLPACK while allowing for memory access coalescing between concurrent threads on GPU. If the slice height C is chosen as 1, the sliced ELLPACK format becomes identical to the CSR format, which has optimal storage efficiency. Padded jagged diagonal storage (pJDS) was proposed in [13]; it involves a row reordering to further increase the storage efficiency for sliced ELLPACK when C is larger than one. However, the row reordering can be detrimental to the accessing pattern of the input vector. Kreutzer et al. [14] proposed the SELL-C-σ format, which limits the reordering scope to σ rows instead of the global matrix. The parameters C and σ define a unified framework for the implementation and performance analysis of all ELLPACK-based formats.

Liu et al. [15] proposed the ELLPACK sparse block format that partitions the matrix by rows and columns into large sparse blocks, thus achieving much better performance than does CSR on the Intel Xeon Phi.

2.6 KNL architecture

The KNL processor has up to 72 cores with multiple versions available containing either 64 or 68 cores. On the 64-core variant (KNL 7230), the chip has 64 cores, which are organized into 32 tiles with each tile consisting of two cores sharing a 1 MB L2 cache. Integrated on package are 16 GB of multichannel DRAM (MCDRAM), while DRAM is off-package and connected by six DDR4 channels. The frequency typically boosts by 0.2 GHz in turbo mode and drops by 0.2 GHz if there is a high proportion of AVX instructions.

MCDRAM can be configured into three modes: flat, cache, and hybrid mode. In flat mode, MCDRAM is exposed by the operating system (OS) as an additional NUMA node. In cache mode, MCDRAM serves as a large, direct-mapped last-level (L3) cache. This is the default mode on Cori and Theta, since it requires no modifications to the application code. The hybrid mode allows part of MCDRAM to be used as cache and the rest to be explicitly controlled by the user.

One of the most notable features of KNL is the introduction of the AVX-512 instruction set. Compared with its predecessor AVX2, it not only promotes the vector length from 256 bits to 512 bits, but also adds new capabilities such as more efficient math functions, support for new scalar data types, and more efficient scatter-gather. AVX, AVX2, and AVX-512 can be mixed without performance penalties. On KNL, AVX and AVX2 instructions operate on the lower 128 or 256 bits of the 512-bit ZMM registers.

Figure 4 shows the STREAM benchmark result for one 68-core, 7250 KNL node on NERSC's CORI supercomputer. It plots sustainable memory bandwidth versus the number of MPI processes. MCDRAM memory bandwidth in flat mode scales to almost 500

GB/s, which is comparable to practical memory bandwidth delivered by recent GPUs. However, high process counts and proper vectorization are needed to achieve high bandwidth: 58 processes are needed to saturate in flat mode, and 40 processes are needed in cache mode. We note that in flat mode, use of vectorization results in dramatically higher achieved memory bandwidth versus unvectorized code, but in cache mode disabling vectorization only slightly lowers the achieved bandwidth.

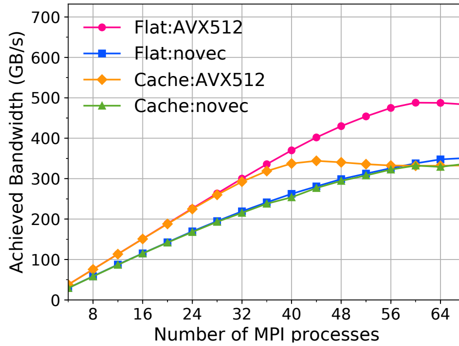

Figure 4: Stream tests on KNL.

3 GENERAL OPTIMIZATION CONSIDERATIONS ON KNL

In the following we summarize code optimization strategies that are particularly effective for KNL. The performance on conventional multicore CPUs typically benefits from these optimizations as well.

3.1 Data alignment

Creating data on aligned boundaries can improve SIMD performance on KNL. If the data in a vectorized loop is not aligned to the cache line size, the compiler has to generate extra code to handle the calculation for the portion of data not beginning at the cache line boundary, resulting in a performance penalty. For example, in Figure 5, given a cache line size of 64 bytes, an array aligned on 16-byte boundary requires special handling at the beginning of the loop, which is called peel code. Aligning the data to 64 bytes can avoid the execution of peel code.

In PETSc, dynamic allocations on the heap are always aligned by some number of bytes set at configuration time, which can be changed with the configuration option `-with-mem-align=<n>`. By default, 16 bytes is used in PETSc. With this default alignment setup, a build of PETSc with AVX-512 instructions enabled (e.g., by passing `-xMIC -AVX512` to the Intel compiler) results in example applications to hang randomly on KNL because of the incorrect alignment. The use of 64-byte alignment fixes the problem and provides better performance because it matches the cache line size.

3.2 Blocking

Blocking is a common optimization technique that exploits inherent data reuse by ensuring that the data remains in cache or registers across multiple uses. When applied to SpMV, it normally involves loop splitting and data rearrangement.

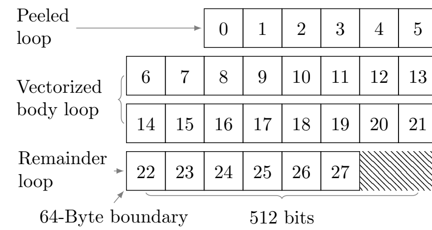

Figure 5: Vectorizing a loop for double precision floating-point computations.

Since the number of registers is relatively small, the goal of register-level blocking is to identify small dense blocks in the matrix so that portion of data in the input vector can be reused and indexing overhead can be reduced. For matrices with natural blocks, for example, PDE problems with multiple degrees of freedom, transforming SpMV into multiplications of small dense blocks and small vector segments is convenient and easy to transform. Otherwise, matrix reorganization or zero padding may become necessary to form small blocks. This will make the code difficult to write and the savings highly dependent on the block size selected. PETSc provides the BAIJ matrix type, which is based on the blocked CSR format [9, 11], for problems with natural blocks; but it does not perform register blocking for general matrices. Saule et al. reported that the use of register blocking for CSR-based kernels has no performance improvement on the first-generation “Knights Corner” Xeon Phi [22]. Thus, in this paper we do not pursue register blocking for the general CSR format.

On KNL, blocking seems to be more difficult because of the large size of the vector registers. Matrices with small natural blocks would need zero padding or masked vector operations, yielding loss in SIMD efficiency.

Cache-level blocking can be a more attractive option because it does not require a block structure to deliver good performance. Row or column partitioning has been often used on both sparse and dense matrices that do not fit into cache.

3.3 Loop remainder vectorization

Significant vectorized loop execution time is often spent in remainder loops. Although the remainder loop can be vectorized by using masked operations on KNL, significant overhead is involved in setting up the mask and executing a separate code path. This can result in low SIMD efficiency; and the vectorized remainder loops can be even slower than the scalar executions, because of the overhead of masked operations and hardware or software padding. Normally the compiler can determine if the remainder should be vectorized based on an estimate of the potential gain. When trip count information for a loop is unavailable, however, it will be difficult for the compiler to make the optimal decision. Moreover, it is not feasible for library developers or users to specify the range of trip counts with `#pragma` because the number may vary significantly across applications.

Whether the loop remainder is executed depends on the vector length, workloads at runtime, and the unroll factor of the loop. For example, given a vector length of 8, when looping over an array of eight-byte data types, any loop with fewer than 8 iterations will be executed in remainder loops.

3.4 Using MCDRAM

MCDRAM can be used in two ways under the flat mode of KNL. One way is to use `numactl`, a command line tool that controls NUMA scheduling or memory placement policy. The other way is to explicitly call functions from memkind, a heap manager library built on top of jemalloc, which provides partitioning of the heap into different "kinds" of memory (normal-bandwidth DRAM vs. high-bandwidth MCDRAM in this case). An advantage of memkind is that the user is free from the burden of tracking which pool of memory an allocation has come from and which deallocator must be used. This greatly simplifies memory management when there is need to use multiple types of memory at the same time (e.g., save checkpoints on DRAM while perform computation using MCDRAM). We have implemented routines to use memkind as an allocator in PETSc, but for the flat mode experiments presented here, we used the `numactl` approach.

4 OPTIMIZATIONS FOR CSR FORMAT

The most effective optimization techniques for a CSR-based SpMV kernel targeting earlier CPUs without wide vector registers are register blocking [12] and cache blocking [18]. As pointed out by Liu et al. in [15], however, register blocking is not appropriate for CPUs with wide SIMD and large register files such as KNC because (i) it is difficult to produce large dense blocks that fit into the registers by reordering sparse matrices and (ii) excessive zero padding has to be used and may result in low SIMD efficiency.

Algorithm 1 Sparse matrix-vector product in CSR (rowptr,colidx,val)

1: **for** $rowid \leftarrow rowstart, rowend$ **do**
2: $\quad idx \leftarrow rowptr[rowid]$
3: $\quad y[rowid] \leftarrow 0$
4: $\quad$ **while** $idx < \lfloor rowptr[rowid+1]/8 \rfloor * 8$ **do**
5: $\quad\quad vec_vals \leftarrow \text{load}(\&val[idx])$
6: $\quad\quad vec_idx \leftarrow \text{load}(\&colidx[idx])$
7: $\quad\quad vec_x \leftarrow \text{gather}(vec_idx, x)$
8: $\quad\quad vec_y \leftarrow \text{fmadd}(vec_vals, vec_x, vec_y)$
9: $\quad\quad idx \leftarrow idx + 8$
10: $\quad$ **end while**
11: $\quad$ **while** $idx < rowptr[rowid] + 1$ **do** ▷ remainder loop
12: $\quad\quad y[rowid] \leftarrow y[rowid] + val[idx] * x[colidx[idx]]$
13: $\quad$ **end while**
14: $\quad y[rowid] \leftarrow y[rowid] + \text{reduce_}add(vec_y)$
15: **end for**

We optimized the SpMV kernel for CSR by vectorizing the inner loop that computes the inner product of one row of the matrix and the input vector x, as shown in Alg. 1. Assuming the floating-point numbers are represented in 64-bit double precision, 8 matrix elements are loaded from the *val* directly into a 512-bit (ZMM) register since they are stored continuously in memory. The corresponding 8 vector elements need to be gathered from x into another ZMM register based on the column indices loaded from *colidx*. A fused multiply-add instruction is applied to multiply the contents of the two vector registers and add the results to vec_y. If the row length is not a multiple of 8, a remainder loop will be left. We vectorize the loop in a similar way only if the length is larger than 2. Because the length is smaller than 8, masked gather and fmadd are used instead.

5 IMPLEMENTATION AND OPTIMIZATION OF SLICED ELLPACK FORMAT

The format we developed in PETSc is based on the sliced ELLPACK, which is similar to the ESB format [15]. As depicted in Figure 6, it can be represented by an array `val` storing the nonzero values and padded zeros, an array `colidx` storing the column indices, an array `rlen` storing the number of nonzeros in each row, and an array storing the beginning position (index in `val`) of each slice. The rows are padded with zero values so that the lengths of the rows in each slice are equal.

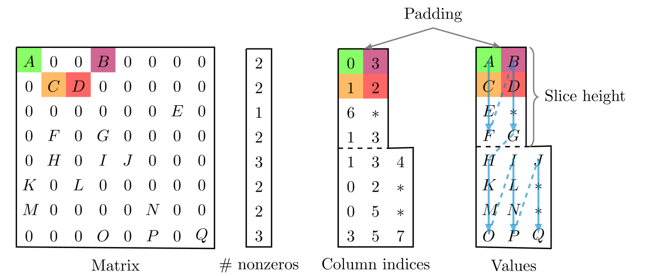

Figure 6: Sliced ELLPACK format.

5.1 Slicing

Slicing is the most important modification to the original ELLPACK format. It not only reduces zero-padding significant but also promotes data locality in accessing the input vector. The lower the slice height, the less zero padding would be required. On GPUs it is preferable to choose the slice height to be multiple of the number of threads per warp or wavefront. In order to effectively utilize vector registers in CPUs, the slice height should be a multiple of the vector length, which is equal to the cache line length of SIMD registers. Therefore, we use a fixed slice height of 8 on KNL for double precision.

5.2 Extra array `rlen`

The array `rlen` stores the actual length of each row. The GPU implementation of SpMV based on ELLPACK-R use it as the bound for the inner loop essentially the same way as the kernel based on sliced ELLPACK. We do not need to use this array explicitly in the SpMV implementation to reduce useless computations as GPU, but it is particularly useful for other functions in PETSc for purposes such as identifying padded zeros, matrix setup, preallocation, and matrix assembly. Also, it is needed by other existing matrix formats in PETSc.

5.3 No bit array

A bit array was used by Liu et al. in the ESB format to provide masks for the vectorization in SpMV so that padded zeros are completely avoided. We prefer not to use the bit array based on the following considerations:

- Most PETSc applications do not have extremely irregular matrices, and slicing can efficiently reduce the number of padded zeros;
- Bit array takes more storage space (about the 1/64 of space needed by the value array) and introduces extra memory footprint in SpMV;
- Not using the padded zeros results in unaligned data access to the matrix value array `val`.
- Masked instructions requires hardware (AVX2) and compiler support. Because we do not rely on masking, also older CPUs with support for AVX can be targeted.

We have implemented two versions with and without the bit array respectively. Not using the bit array leads to about 10% speedup over using the bit array.

5.4 No sorting

While sorting rows by the number of nonzero elements can significantly reduce the padded zeros for irregular matrices (this is the approach used in AIJPERM), we do not think it is ideal to perform sorting in the low-level linear algebra kernels. Instead, the ordering of variables can be better adjusted at a higher level of a PDE solver library, in particular in the grid partitioning stage. As a result, a single reordering of the grid allows for all subsequent simulations using the grid to use the better ordering.

Another potential disadvantage of sorting in low-level linear algebra kernels is the loss of data locality in the input vector after sorting. There are heuristic techniques that limit sorting into a small portion of the matrix to protect locality to some extent. However, the difficulty of finding an optimal range for sorting originates from the fact that these kernels are agnostic to the domain information represented by the input vector.

5.5 SpMV based on sliced ELLPACK

Algorithm2 shows the serial SpMV kernel based on the sliced ELLPACK format. The matrix-vector multiplication is performed slice-wise. At each iteration of the outer loop, one slice of the matrix is used to update eight contiguous elements in the output vector. The values and column indices of the elements in a slice are loaded from memory column by column, which is exactly the same order as they are stored physically. The column indices also determine the positions of the sparse data in x that are to be multiplied with the matrix values. We use gather instructions to collect the data and fused multiply-add instructions to accumulate the product for 8 contiguous rows. The product is stored into y after the slice is processed.

Algorithm 2 Sparse matrix-vector product for the sliced ELLPACK format

```
1:  for sliceid ← slicestart, sliceend do
2:      setzero(vec_y)
3:      idx ← sliceptr[sliceid]
4:      while idx < sliceptr[sliceid + 1] do
5:          vec_vals ← load(&val[idx])
6:          vec_idx ← load(&colidx[idx])
7:          vec_x ← gather(vec_idx, x)
8:          vec_y ← fmadd(vec_vals, vec_x, vec_y)
9:          idx ← idx + 8
10:     end while
11:     store(vec_y, &y[sliceid * 8])
12: end for
```

With proper padding, we can execute almost all the vector instructions without masks. If the last slice has fewer than eight rows, we pad additional rows of zeros to ensure that the number of elements in each slice is multiple of 8, and masked store operations are needed when the last slice is vectorized. The padding not only results in aligned access to each slice when the matrix is aligned but also eliminates the need to process a remainder. We also note that the column indices of the padded elements are copied from those of local nonzero elements to avoid triggering interprocess communication.

In our implementation we have manually unrolled the outer loop and performed a prefetch operation before the inner loop starts. However, these classic optimization techniques do not affect the performance significantly.

We have implemented the kernel for the AVX, AVX2, and AVX-512 instruction sets, respectively. The AVX- and AVX2-instructions include all the vector operations needed in Alg. 2, but the vector length is half of the AVX-512 vector length. Therefore, the total number of instructions executed is doubled with AVX2. AVX supports the same vector length as AVX2, but it does not have fmadd and gather instructions. We use separate multiply and add instead of fmadd. The gather operation is replaced with load and insert operations. We use two SSE2 load instructions to load two 64-bit floating point values into a packed vector and then insert two packed 128-bit vectors to form a 256-bit AVX vector.

6 PERFORMANCE EVALUATION

Since SpMV is known to be memory bandwidth bound on most modern computer architectures [9], it is reasonable to look into the memory bandwidth usage when comparing the performance for different SpMV kernels. The estimated memory bandwidth is the minimum memory traffic required just to transfer the matrix values and vector values. We assume that a floating-point data element takes 8 bytes and an integer value in the index array takes 4 bytes. For a sparse matrix with m rows, n columns and nnz nonzeros, the SpMV kernel based on CSR needs to move at least $12nnz + 24m + 8n$ bytes data from memory. The first term accounts for the memory traffic for accessing matrix values and column indices. Accessing the vectors x and y takes $8m$ and $8n$ bytes, respectively, excluding any redundant memory accesses; and the array that contains the starting index of each row takes $8m$ bytes. Both the diagonal and

off-diagonal part have this array. In contrast, the sliced ELLPACK-based SpMV kernel takes $12nnz + 10m + 8n$ bytes since the matrix can be traversed slicewise and storing the starting positions of all slices takes $m/8$ integer values. Extra memory overhead contributed by padded zeros are not counted in order to eliminate artifacts due to implementation that may boost the performance.

7 RESULTS

We evaluate the performance of the new kernels for a classical reaction-diffusion system that simulates the interaction of two chemical species on a 2D rectangular grid using the Gray-Scott model [20].

$$\begin{aligned} \frac{d\mathbf{u}}{dt} &= D_1\nabla^2\mathbf{u} - \mathbf{u}\mathbf{v}^2 + \gamma(1-\mathbf{u}) \\ \frac{d\mathbf{v}}{dt} &= D_2\nabla^2\mathbf{v} + \mathbf{u}\mathbf{v}^2 - (\gamma+\kappa)\mathbf{v} \end{aligned} \tag{1}$$

The PDEs are discretized with central finite differences by using a 5-point stencil. The parameter settings follows page 21 of the book by Hundsdorfer and Verwer [10], except that periodic boundary conditions are used instead of homogeneous Neumann conditions for simplicity. We use the Crank-Nicolson scheme with a fixed step size of 1 for time integration. In the experiments 20 time steps are taken on a single compute node, while 5 time steps are used for large-scale experiments on multiple nodes.

At each time step, a nonlinear system is solved with Newton's method. Because of the nonlinear reaction term that couples $\mathbf{u}$ and $\mathbf{v}$, the Jacobian matrix needs to be updated at each Newton iteration. The Jacobian evaluation and its multiplication with input vectors dominate the simulation, accounting for about half of the total running time.

The 5-point stencil Laplacian operator results in a banded sparse matrix in the diagonal block. The off-diagonal block contains just a few nonzeros due to the boundary condition. Since each grid point has two degrees of freedom, the matrix consists of small 2×2 blocks. Each row has 10 elements. When represented in the sliced ELLPACK format, there are very few padded zeros.

The linear system is solved with the GMRES Krylov subspace method. Since the problem is diffusion dominated, we use a multigrid preconditioner to accelerate convergence and avoid the typical increase in the number of iterations as the grid is refined.

The source code for our implementation and the test problem[1] is distributed in PETSc [1] version 3.9 and later. The log files that are used to generate the figures in this paper are also publicly available [2]. PETSc was built with Intel MKL (version 2018 update 1) BLAS and LAPACK. The MKL SpMV kernel used for comparison is run with the PETSc option `-mat_aijmkl_no_spmv2`, which disables the inspector-executor optimization. The experiments on KNL are performed on Argonne's Theta supercomputer [19], which consists of 4,392 64-core 7230 KNL nodes. All the experiments are carried out in quadrant cluster mode, in which the tiles on the chip are divided into four quadrants and memory addresses associated with a quadrant are mapped only to local tag directories in that quadrant.

[1] In the source, see `src/ts/examples/tutorials/advection-diffusion/ex5adj.c`.
[2] https://bitbucket.org/caidao22/petsclogs. The log files contain configuration options, command line options used to run the tests and profiling details.

7.1 Out-of-box baseline performance

We run the simulations with three different grid resolutions using flat mode and cache mode. By default, the matrix type in PETSc is AIJ, which is the compressed sparse row format. As shown in Figure 7, the performance is insensitive to the grid size for this example while the memory usage does not exceed the limit of MCDRAM capacity. This can be attributed to the sparsity pattern of the matrices. Note that the coarsening process of the multigrid preconditioner results in matrices of different dimension, and for each level matrix-vector products need to be performed. Changing the resolution or the number of levels leads to variations in the number of rows for these sparse matrices, but the number of nonzeros per row, which is determined by the spatial discretization scheme, remains constant. Therefore, it is reasonable to choose a single resolution for the performance comparisons.

When using 16 or 32 processes, there is almost no difference in flop rates between using the MCDRAM or DRAM. The gap becomes noticeable only when all the cores have been filled. The reason is that the memory bandwidth saturates more quickly with DRAM than with MCDRAM as the number of processes increases, and the wide vectorization capability of KNL makes SpMV more bandwidth-hungry. In addition, cache mode yields slightly lower performance than does flat mode, which is consistent with the STREAM benchmark results (Figure 4).

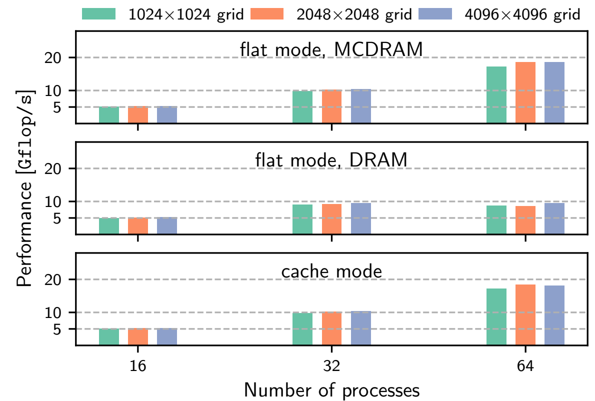

Figure 7: Baseline out of box SpMV performance using CSR for various grid sizes.

7.2 Single-node performance

To evaluate single-node performance, we pick a grid size of 2048×2048, which ensures all the data can fit into the MCDRAM during the simulation. We use three levels of V-cycle multigrid. The smoothers and coarse-level solvers are set to use the Jacobi preconditioner so that the algorithm relies heavily on matrix-vector multiplications. The corresponding PETSc options are listed below:

```
-pc_type mg
-pc_mg_levels 3
-mg_levels_pc_type jacobi
-mg_coarse_pc_type jacobi
```

Figure 8 shows the SpMV performance of different matrix types for a varying number of MPI ranks used. The AVX-512 version clearly outperforms all other kernels and is on average twofold faster than the baseline CSR. The AVX and AVX2 versions of the sliced ELLPACK storage have a speedup of 1.8 and 1.7 over the baseline CSR, respectively. The Intel MKL library performs slightly worse than the baseline CSR kernel, which is automatically optimized by the compiler. This performance indicates that PETSc's default implementation is competitive with the implementation in MKL. CSR with permutation (AIJPERM in PETSc) does not yield any improvement over the CSR baseline. But the performance of CSR-based kernel increases by 54% after being manually optimized by using AVX-512 intrinsics. This indicates that the compiler's ability to automatically vectorize SpMV with the AVX-512 instructions is far from satisfactory.

An interesting observation is that using AVX2 instructions for CSR leads to a regression in performance compared with the AVX version on KNL, while the AVX and AVX2 implementations for SELL are roughly comparable with each other. The AVX2 implementations use gather and fused multiply-add (FMA) instructions that do not exist in AVX. The reason for the performance regression is unclear, but we speculate that the use of separate multiply and add instructions in the AVX version may be beneficial because the multiplication in the ith iteration can be executed independently of the addition in iteration $i-1$, whereas in the AVX2 version the FMA in iteration i cannot begin before the FMA in iteration $i-1$.

All the formats investigated demonstrate good strong scalability up to 64 cores; see Figure 8. Therefore, when running large-scale applications on multiple KNL nodes, one should use all the physical cores and pin one MPI process to one core for best performance.

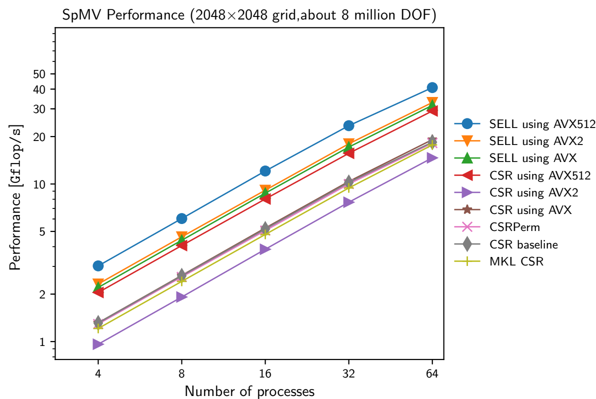

Figure 8: Comparison of various matrix formats on a single KNL node. Each MPI process is affined to one physical core of the processor.

Figure 9 shows a roofline analysis [8, 16], generated by using the Empirical Roofline Tool from LBNL, for the best performance (with the full 64 MPI ranks) achieved by each matrix type. The arithmetic intensity of the SpMV kernel is around 0.132 according to the analysis in Sec. 6. We can see that the AVX-512 version of the sliced ELLPACK SpMV kernel has pushed the baseline performance close to the MCDRAM roofline, which marks the ideal GFLOPs/sec when the MCDRAM bandwidth is saturated.

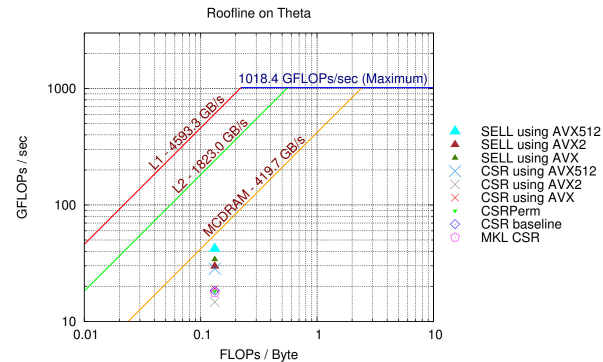

Figure 9: Roofline analysis of the SpMV kernel on KNL.

7.3 Multinode performance

For large-scale tests, we focus on the comparison between the fastest sliced ELLPACK implementation and the CSR baseline (PETSc default). The grid size is increased to $16,384 \times 16,384$, which is close to the largest case that does not require 64-bit integers for indexing. Also, the number of levels of the multigrid preconditioner is set to be 6 so that the problem size is small enough at the coarse level. As can be seen from Figure 10, sliced ELLPACK gives an approximately twofold speedup over CSR for the SpMV kernel when running in cache mode and flat mode. The savings in SpMV translate directly into significant drops in the total wall time because the portion for other parts of the code remain almost the same for the two matrix formats. We note that the changes in the matrix representation result in implementation differences for certain matrix operations such as setting the nonzero entries and assembling the matrix. The corresponding routines for these operations in PETSc are executed every time the Jacobian matrix is updated.

When the KNL nodes are configured to flat mode and the tests use only DRAM, there is just marginal improvement in the SpMV performance using sliced ELLPACK instead of CSR. This indicates together with the roofline plot in Figure 9 that efficient vectorization itself does not translate into good SpMV performance without the support of high-bandwidth memory.

7.4 Performance on other Xeon processors

Vectorization and multicore parallelism are also a feature of recent generations of the standard Intel Xeon architectures such as Haswell, Broadwell, and Skylake. We compare the performance of our optimized kernels on KNL with three recent Xeon processors, the specifications of which are listed in Table 1. All runs are performed by using all the available physical cores with one MPI process pinned to one core.

The results in Figure 11 show only marginal improvement for sliced ELLPACK over CSR on standard Xeon platforms, but significant gains on KNL. These results are partially due to the dramatically improved memory bandwidth of KNL, which is about 4-6 times larger than do the other Xeon processors. Intel MKL is about

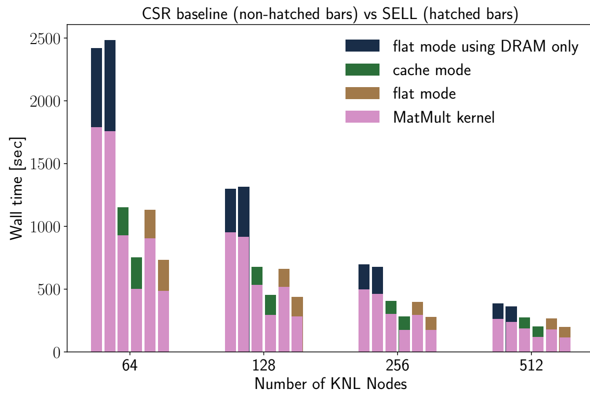

Figure 10: SpMV performance on the supercomputer Theta.

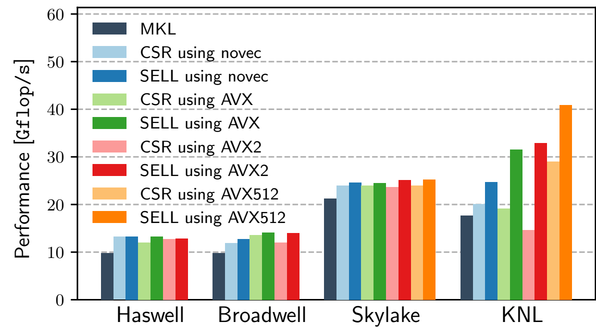

Figure 11: SpMV performance on different Xeon processors.

Table 1: Overview of Intel processors used for evaluating SpMV performance.

	# of Cores	Processor Base Frequency (Turbo)	L3 Cache	Max DDR4 Bandwidth	HBM Bandwidth
KNL 7230	64	1.3(1.5) GHz		115.2 GB/s	>400 GB/s
Broadwell E5-2699v4	22	2.2(3.6) GHz	55 MB	76.8 GB/s	
Haswell E5-2699v3	18	2.3(2.6) GHz	45 MB	68 GB/s	
Skylake 8180m	28	2.5(3.6) GHz	38.5 MB	119.2 GB/s	

10 to 20 percent slower on standard Xeons as well as on KNL when compared with the default CSR implementation. Moreover, Figure 11 shows how the performance of each format is impacted by the evolving Intel architectures. Specifically, sliced ELLPACK performs the best on KNL and its performance increases as wider SIMD instructions are used (compare AVX-512 with AVX). But this trend is not evident for CSR. The AVX-512 version of CSR works better on KNL than on any other platform; however, the best performance of AVX/AVX2 versions of CSR is found on Skylake.

We can see that Skylake gets about twice the performance of Broadwell and Haswell. Skylake supports six memory channels per socket, compared with four per socket for Haswell and Broadwell, yielding significantly higher memory bandwidth. This higher bandwidth largely explains the improved performance on Skylake—the core frequencies of the three Xeon processors are close to each other, and the Skylake processor actually has less L3 cache than the other two processors have. We argue that vectorization becomes important on KNL because the SpMV kernel moves from a memory-bound to compute-bound regime as the memory bandwidth increases. Although the Skylake processor features fairly high memory bandwidth itself, the individual cores are much more powerful than KNL cores, and the balance between core performance and memory bandwidth is such that the SpMV stays in a compute-bound regime.

8 CONCLUSION

In this paper, we share our experience performing analysis and optimization of PETSc's parallel SpMV kernel through vectorization. By manually rewriting the code with AVX-512 intrinsics, we have achieved a good speedup on KNL in comparison with the existing CSR format.

However, CSR is not the optimal choice for matrices whose number of nonzeros per row is either small or not a multiple of the length of the CPU vector register, which are common in the PDE regime. To address this situation, we have developed a new PETSc matrix type based on the sliced ELLPACK format, and we investigated the potential for preconditioned iterative solvers to benefit from the accelerated kernels based on the new format. The potential has been demonstrated with a representative reaction-diffusion PDE using much of the solver hierarchy in PETSc. We have also compared the sliced ELLPACK format with several CSR variants including the PETSc default version, a permutation based version, and a version provided by Intel's MKL library. For this particular test, the sliced ELLPACK-based SpMV implementation delivers a twofold speedup over the CSR baseline on KNL, and the hand-optimized CSR kernel performs 54% better than the compiler-optimized CSR kernel. Furthermore, we show there is no noticeable performance penalty in other core operations needed by a practical PDE solver. Similar optimizations for Xeon CPUs did not result in a significant performance gain, indicating that explicit vectorization is not yet a necessity for obtaining peak performance for SpMV on those architectures.

In future work we will investigate further optimization opportunities for the sliced ELLPACK format for other kernels such as (possibly incomplete) LU decomposition and triangular solves for sliced ELLPACK in order to make it usable with more preconditioner choices. It may be particularly challenging to balance the higher generality of the CSR format with the SpMV-centric nature of the sliced ELLPACK format in a complicated solver setup.

Our results also indicate that modern compilers are still not able to generate optimal assembly code for simple computational kernels like SpMV, and extracting good performance for vector processors still requires significant effort and fine restructuring of the code. The optimized implementation of sparse matrix kernels may have to be modified with the CPU instruction sets evolving and the hardware features being continuously strengthened, but we believe the essential optimization strategies such as densifying the sparse matrices and improving the loop remainder vectorization efficiency will remain relevant in the many-core era.

ACKNOWLEDGMENTS

We thank Vamsi Sripathi of Intel Corporation for several useful discussions about Intel AVX, AVX2, and AVX-512 intrinsics functions, and details of Xeon and Xeon Phi processor microarchitecture. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357 and the Exascale Computing Project (Contract No. 17-SC-20-SC). This material is also based upon work supported by the VSC Research Center funded by the Austrian Federal Ministry of Science, Research and Economy (bmwfw). This research used resources (the Intel Xeon Phi 7250 "Knights Landing" nodes of the Cori Cray XC40 system) of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and resources of the Argonne Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2017. *PETSc Users Manual.* Technical Report ANL-95/11 - Revision 3.8. Argonne National Laboratory. http://www.mcs.anl.gov/petsc

[2] Taylor Barnes, Brandon Cook, Jack Deslippe, Douglas Doerfler, Brian Friesen, Yun He, Thorsten Kurth, Tuomas Koskela, Mathieu Lobet, Tareq Malas, Leonid Oliker, Andrey Ovsyannikov, Abhinav Sarje, Jean Luc Vay, Henri Vincenti, Samuel Williams, Pierre Carrier, Nathan Wichmann, Marcus Wagner, Paul Kent, Christopher Kerr, and John Dennis. 2017. Evaluating and optimizing the NERSC workload on Knights Landing. In *Proceedings of PMBS 2016: 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, St.* 43–53. https://doi.org/10.1109/PMBS.2016.010

[3] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09.* Bell2009, 1. https://doi.org/10.1145/1654059.1654078

[4] Wei Cao, Yao Lu, Zongzhe Li, Yongxian Wang, and Zhenghua Wang. 2010. Implementing Sparse Matrix-Vector multiplication using CUDA based on a hybrid sparse matrix format. In *ICCASM 2010 - 2010 International Conference on Computer Application and System Modeling, Proceedings*, Vol. 11. IEEE, V11–161–V11–165. https://doi.org/10.1109/ICCASM.2010.5623237

[5] Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM SIGPLAN Notices* 45, 5 (may 2010), 115. https://doi.org/10.1145/1837853.1693471

[6] Edmond Chow. 2001. Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners with a Priori Sparsity Patterns. *The International Journal of High Performance Computing Applications* 15, 1 (feb 2001), 56–74. https://doi.org/10.1177/109434200101500106

[7] Eduardo F. D'Azevedo, Mark R. Fahe, and Richard T. Mills. 2005. Vectorized sparse matrix multiply for compressed row storage format. *[ICCS'05] Computational Science–ICCS 2005* 3514/2005 (2005), 99–106. https://doi.org/10.1007/11428831_13

[8] Douglas Doerfler, Jack Deslippe, Samuel Williams, Leonid Oliker, Brandon Cook, Thorsten Kurth, Mathieu Lobet, Tareq Malas, Jean Luc Vay, and Henri Vincenti. 2016. Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 9945 LNCS. 339–353. https://doi.org/10.1007/978-3-319-46079-6_24

[9] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2009. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *Journal of Supercomputing* 50, 1 (2009), 36–77. https://doi.org/10.1007/s11227-008-0251-8

[10] Willem Hundsdorfer and Jan G. Verwer. 2003. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations.* Number 33 in Springer series in computational mathematics. Springer.

[11] Eun-Jin Im and Katherine Yelick. 1999. Optimizing Sparse Matrix Vector Multiplication on SMPs. In *In Ninth SIAM Conference on Parallel Processing for Scientific Computing.*

[12] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158. https://doi.org/10.1177/1094342004041296

[13] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R. Bishop. 2012. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012.* 1696–1702. https://doi.org/10.1109/IPDPSW.2012.211

[14] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423. https://doi.org/10.1137/130930352 arXiv:1307.6209

[15] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS '13* (2013), 273. https://doi.org/10.1145/2464996.2465013

[16] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. 2015. Roofline model toolkit: A practical tool for architectural and program analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 8966. 129–148. https://doi.org/10.1007/978-3-319-17248-4_7

[17] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 5952 LNCS. 111–125. https://doi.org/10.1007/978-3-642-11515-8_10

[18] Rajesh Nishtala, Richard W. Vuduc, James W. Demmel, and Katherine A. Yelick. 2007. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communications and Computing* 18, 3 (2007), 297–311. https://doi.org/10.1007/s00200-007-0038-9

[19] Scott Parker, Vitali Morozov, Sudheer Chunduri, Kevin Harms, Chris Knight, and Kalyan Kumaran. 2017. *Early Evaluation of the Cray XC40 Xeon Phi System 'Theta' at Argonne.* Technical Report. Argonne National Laboratory.

[20] John E. Pearson. 1993. Complex Patterns in a Simple System. *Science* 261, 5118 (1993), 189–192.

[21] Ali Pinar and Michael T. Heath. 1999. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '99.* ACM Press, New York, New York, USA, 30–es. https://doi.org/10.1145/331532.331562

[22] Erik Saule, Kamer Kaya, and Umit V. Catalyurek. 2013. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 8384 LNCS. 559–570. https://doi.org/10.1007/978-3-642-55224-3_52 arXiv:arXiv:1302.1078v1

[23] Francisco Vázquez, José-Jesús Fernández, and Ester M. Garzón. 2011. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency Computation Practice and Experience* 23, 8 (2011), 815–826. https://doi.org/10.1002/cpe.1658 arXiv:arXiv:1302.5679v1

[24] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178–194. https://doi.org/10.1016/j.parco.2008.12.006

[25] Jianfei Zhang and Lei Zhang. 2013. Efficient CUDA polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems. *Mathematical Problems in Engineering* 2013 (2013). https://doi.org/10.1155/2013/398438

[26] Cong Zheng, Shuo Gu, Tong Xiang Gu, Bing Yang, and Xing Ping Liu. 2014. BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs. *J. Parallel and Distrib. Comput.* 74, 7 (jul 2014), 2639–2647. https://doi.org/10.1016/j.jpdc.2014.03.002