Characterization and optimization of sparse computations on Intel Xeon Phi

A. Elafroua, G. Goumas*

*National Technical University of Athens

Abstract

In this paper, we propose a lightweight optimization methodology for the ubiquitous sparse matrix-vector multiplication (SpMV) kernel for the Intel Xeon Phi manycore processors. The large number of cores in this platform overly exposes inherent structural weaknesses of different sparse matrices, intensifying performance issues beyond the traditionally reported memory bandwidth limitation. We, thus, advocate an input-adaptive optimization approach and present a method that identifies the major performance bottleneck of the kernel for every instance of the problem and selects a suitable optimization to tackle it. We describe two models for identifying the bottleneck: our first model requires performance bounds to be determined for the input matrix during an online profiling phase, while our second model only uses comprehensive structural features of the sparse matrix. Our optimizations are based on the widely used Compressed Sparse Row (CSR) storage format and have low preprocessing overheads, making our overall approach practical even in the context of iterative solvers that converge in a small number of iterations. We evaluate our methodology on the Intel Xeon Phi co-processor, codename Knights Corner (KNC), and demonstrate that it is able to distinguish and appropriately optimize the great majority of matrices in a large and diverse test suite, leading to a significant speedup of 2.2× on average over the Intel MKL library.

1. Introduction

The ubiquitous sparse matrix-vector multiplication (SpMV) kernel is a fundamental building block of popular iterative methods for the solution of sparse linear systems (Ax = b), and the approximation of eigenvalues and eigenvectors of sparse matrices (Ax = λx). Such problems arise in a diverse set of application domains, including large-scale simulations of physical processes using multi-physics and multi-disciplinary approaches, information retrieval, medical imaging, economic modeling and others. Optimizing SpMV has always been a challenging task due to a number of inherent performance limitations, as a result of the algorithmic nature of the kernel, the employed sparse matrix storage format and the sparsity pattern of the matrix. SpMV is characterized by a very low flop:byte ratio, since the kernel performs O(\(NNZ\)) operations on O(\(NNZ + N\)) amount of data, indirect memory references as a result of storing the matrix in a compressed format and irregular memory accesses to the source vector due to sparsity.

On multicore architectures, SpMV has typically been reported to be a memory bandwidth-bound kernel for the majority of sparse matrices [1]. Consequently, most optimization efforts proposed in the literature over the past few decades have focused on reducing traffic between caches and main memory, primarily by compressing the memory footprint of the matrix [2] - [12]. A subset of these efforts has addressed less prominent performance issues that were usually related to a minority of sparsity patterns [13], [14], [15]. However, with the advent of

* Corresponding author. E-mail address: goumas@cslab.ece.ntua.gr
modern manycore architectures, including GPUs and the Intel Xeon Phi co-processor, as well as the use of dual-processor compute nodes featuring high core numbers in most supercomputers, the performance landscape has changed. Architectural diversity with regard to the number and complexity of cores, the use of SMT, the memory hierarchy (depth and capacity), and the available memory bandwidth, intensifies different weaknesses of the kernel, leading to bottleneck diversity. For example, the higher number of cores and increased bandwidth in some systems makes SpMV performance more sensitive to workload imbalance among threads and less sensitive to the working set size. We anticipate this trend will become even more relevant in future exascale systems that plan to adopt manycore architectures as their main processors, e.g., the next generation of Intel Xeon Phi, codename Knights Landing.

Since there is no universal optimization that can address all potential performance issues, as they are both architecture- and input-dependent, we have come to believe that the next solid step for improving SpMV performance on current and future platforms no longer relies exclusively on designing new and expensive optimizations, but, more importantly, on applying an optimization whenever it can be effective, i.e., mitigating the challenge of developing new optimizations by selecting an appropriate optimization for the target matrix and architecture. Towards this direction, we propose a methodology to automatically select an efficient optimization for SpMV that is both application- and architecture-aware. Our approach seeks to provide a) performance stability, i.e., optimization selection should provide performance improvements for all sparse matrices, and b) low overhead, i.e., the online overhead of the optimization process should be low in order for it to be applicable in iterative solvers than may require dozens to thousands of iterations to converge. We especially focus on designing a decision-making component that is more lightweight than a trivial optimizer that simply runs all optimizations and selects the best for the target matrix.

In order to provide performance stability, our methodology attempts to detect the major performance bottleneck of SpMV for the input matrix. We formulate the decision making as a classification problem in Section 3-A, assuming classes represent performance bottlenecks. We then develop a profiling-based classifier in Section 3-C, that relies on performance bounds to classify the matrix. As the online-profiling phase has a non-negligible cost, which may outweigh the optimization benefit, we go one step beyond, and propose in Section 3-D a classifier that relies only on structural features of the input matrix, avoiding any online, profiling-based information. This feature-based classifier is pre-trained during an offline stage with the use of machine learning techniques, and only performs feature extraction on-the-fly. The runtime overhead of this classifier is very low, equivalent to only a couple of dozen SpMV operations, making it extremely lightweight.

Once the prevailing performance bottleneck of a matrix has been detected, we accordingly apply an optimization that could tackle it. We employ a simple and easy-to-implement set of optimizations, presented in Section 3-E, based on the general-purpose Compressed Sparse Row (CSR) sparse matrix storage format, among which some are found in the literature, while others are new and target specific sparsity patterns that we distinguish using structural matrix features. Our approach, however, is compatible to a plethora of other optimizations. Again here, the idea is to resort to optimizations that require minimal preprocessing time, further contributing to our objective for a lightweight scheme.

We have tested our method on the Intel Xeon Phi manycore co-processor for a wide and diverse set of matrices. Our experimental results, presented in Section 4, demonstrate that our approach is able to distinguish and appropriately optimize the majority of matrices, leading to a significant speedup of 2.2× on average over the Intel MKL library. The work presented here has potential links to CoEs MAX, BioExcel, ESIWACE and EoCoE. Also, it can be integrated in widely used sparse solver libraries, including the PETSc [34], [35] and Trilinos [36] scientific toolkits.

2. Background

In order to avoid the extra computation and storage overheads imposed by the large majority of zero elements contained in a sparse matrix, it has been the norm to store the nonzero elements of the matrix contiguously in memory and employ auxiliary data structures for their proper traversal. The most widely used general-purpose sparse matrix storage format, namely the Compressed Sparse Row (CSR) format [16], uses a row pointer structure to index the start of each row within the array of nonzero elements, and a column index structure to store the column of each nonzero element. An example of this format is given in Figure 1.
Figure 1 The Compressed Sparse Row (CSR) sparse matrix storage format

The SpMV kernel using this format is given in Figure 2. Examining Figure 2, we notice multiple performance issues for SpMV:

- **low operational intensity** The kernel performs $O(\text{NNZ})$ floating point operations on $O(\text{NNZ} + N)$ amount of data, leading to a flop:byte ratio of less than 1. According to the Roofline performance model [17], computational kernels with low operational intensities tend to be memory-bound.

- **indirect memory references** This is the most apparent implication of sparsity. Since we only want to store the nonzero elements of the matrix, we need auxiliary indexing structures to be able to access them from memory. For the CSR format we use the rowptr and colind data structures. Indexing, however, introduces additional load operations, traffic for the memory subsystem, and cache interference [3].

- **irregular memory accesses to input vector** Access to vector $x$ is irregular and depends on the sparsity pattern of the matrix. This fact complicates the process of exploiting any spatial or temporal reuse with regard to vector $x$ [18].

- **loop overheads** Many sparse matrices contain a large number of short rows. This fact may degrade performance due to the overhead incurred by the small trip count of the inner loop [14].

- **uneven workload** Many sparse matrices are characterized by rows with highly uneven lengths. This fact may introduce workload imbalance for a large number of threads, if the matrix is distributed in a row-wise manner.

![Sparse Matrix](image)

**Figure 2** SpMV implementation using the CSR format

3. **Optimization Selection Methodology**

Depending on the sparsity pattern of the matrix and the underlying architecture, a suitable optimization for SpMV may vary due to varying performance issues. Thus, SpMV could benefit from an optimization selection process. The benefits of such an approach are twofold: firstly, performance can be optimized for all problem instances, and, secondly, we can leverage previous research on SpMV that has generally focused on different instances of the problem.
A. Formulation as a Classification Problem

The optimization selection problem can be solved in various ways. One could simply take an empirical approach: measure how different optimizations work for a particular matrix on the target machine and then apply the most efficient optimization on future runs of the same matrix. However, this would incur a substantial overhead to apply all candidate optimizations, some of them requiring significant preprocessing time. We propose a more elegant and lightweight approach to solve the optimization selection problem. We formulate it as a classification problem, by assuming that every matrix belongs to a single class, representing its major performance bottleneck. For every class, we assign a corresponding optimization that attempts to tackle the specific bottleneck. Given an input sparse matrix, its class is identified and the corresponding optimization is applied. In this context, we define the following classes:

- **MB**: This class includes matrices that have saturated the available main memory (DRAM) bandwidth. This is the dominating class for SpMV on modern multicore architectures due to its very low flop:byte ratio.
- **CML**: This class refers to matrices that suffer from excessive Last Level Cache (LLC) misses and can, therefore, be limited by cache miss latencies. The source of these misses, indicated in Section 2, is the irregular memory access pattern to the input vector, which cannot be detected by hardware prefetching mechanisms available in current architectures.
- **IMB**: This class appears mostly when a large number of threads is used, exposing highly uneven row lengths in the matrix or regions with completely different sparsity patterns, resulting respectively in workload imbalance or computational unevenness.
- **CMP**: This class includes matrices that suffer from computational bottlenecks. Such matrices are mostly matrices that fit in the system’s caches or have very short rows.

The above classes are quite generic, as they serve our initial goal, which is to categorize a matrix based on its prevailing performance bottleneck. Each class covers a wide variety of matrices with different structural characteristics.

Instead of defining our own classes we could have used cluster analysis to discover groups of matrices that are similar in some manner. However, it seemed more intuitive to leverage our own experience and the extensive research that has been realized over the past few decades on SpMV optimization, and define our own classes. A “blind” machine learning approach is not necessary in this case, since the performance bottlenecks of the kernel have already been identified, granting us a basic understanding of the problem. Another option would be to directly define optimizations as classes. Every class would simply represent the best optimization for its group of matrices. However, in this scenario, building a classifier would require associating features of a matrix to every optimization in our search space instead of every performance bottleneck, which could be more unclear, as there are matrices that have some common features, but exhibit different bottlenecks. Also, this approach cannot lead to a cross-platform decision making process, as it is optimization-dependent and optimizations can be completely different from one hardware platform to another. Finally, our classification approach decouples bottleneck identification from the application of optimizations, and, hence, could serve as a tool for performance exploration of the SpMV kernel on new platforms.

B. Performance Bounds

In this section, we perform a thorough bound and bottleneck analysis for the SpMV kernel. This analysis is inspired by “bounding techniques” described in [19] as a tool that can “provide valuable insight into the primary factors affecting the performance of computer systems”, and “estimate the potential performance gain of alternative upgrades to existing systems”. We select CSR as our baseline, as it the most widely-used format in real-world applications. Our analysis, however, can be applied to any format.

To characterize the input matrix, we assume that it belongs to any of the classes defined in Section III-A, and then derive an upper bound for each class. By properly comparing these bounds with the CSR performance, we will classify the matrix as described in Section 3-C. The per-class upper bounds represent the maximum performance that can be achieved using the CSR format by addressing the corresponding performance bottleneck. For completeness, we also define a generic upper bound that is not tied to a specific format, and represents peak SpMV performance. In the following, NNZ refers to the number of nonzero elements in a sparse matrix.
• **MB** For matrices belonging to this class, the available bandwidth between the caches and main memory is saturated. Thus, an upper bound for performance can be derived by assuming maximum sustainable DRAM bandwidth during SpMV execution, as:

\[
P_{MB} = \frac{2 \cdot NNZ}{M_{min}} B_{max}
\]

where \(B_{max}\) is the maximum sustainable DRAM bandwidth of the system, that can be measured by STREAM [20] or a similar benchmark, and \(M_{min}\) is the minimum memory traffic generated when the matrix is stored in the CSR format. Note that compulsory misses set the minimum memory traffic and, hence, \(M_{min} = \text{size}_{CSR} + \text{size}_x + \text{size}_y\). This bound is basically an upper bound for SpMV performance using the CSR format.

• **CML** For this class, we can derive an upper bound \(P_{CML}\) by running a modified SpMV kernel, where irregular accesses to the input vector are completely eliminated by setting the column indices of all nonzero elements to zero. More specifically, we set all entries of the colind array of the CSR format to zero.

• **IMB** For matrices in this class, performance can be maximized by perfectly balancing the workload. Thus, an upper bound for this class can be estimated by using the median execution time \(t_{median}\) of the threads as:

\[
P_{IMB} = \frac{2 \cdot NNZ}{t_{median}}
\]

We use the median instead of the average, as we require reduced importance to be attached to outliers.

• **CMP** For matrices that are limited by computational bottlenecks, we can define a very loose upper bound \(P_{CMP}\) by running a modified SpMV kernel where we completely eliminate indirect memory references, resulting in unit-stride accesses only. More specifically, we no longer use colind to index vector \(x\), but always access \(x[0]\). Intuitively, this is equivalent to converting the sparse kernel to a dense one.

A generic upper bound on SpMV performance can be determined by assuming maximum memory bandwidth and minimum memory traffic as

\[
P_{PEAK} = \frac{2 \cdot NNZ}{M_{min}} B_{max}
\]

where \(B_{max}\) is the measured maximum sustainable DRAM bandwidth of the system, as before. In order to define \(M_{min}\) in this case, we assume we can only compress the indexing information of a sparse matrix (not the values). Thus, we can define memory traffic as \(M = \text{size}_{indexing} + \text{size}_{values} + \text{size}_x + \text{size}_y\), and minimize it by completely eliminating the indexing structures, i.e., setting \(\text{size}_{indexing} = 0\).

Figure 3 shows the performance of a baseline CSR SpMV kernel (no vectorization and no prefetching applied), along with the previously defined per-class and generic upper bounds, for a total of 60 matrices from the University of Florida Sparse Matrix Collection [21]. Examining this figure in detail, we distinguish the following cases:

• **PCSR ≈ PCML** Performance did not improve by completely eliminating irregular memory accesses, hence SpMV does not seem to suffer from excessive cache misses.

• **PCSR ≈ P_{IMB}** This implies that all threads had an execution time close to \(t_{median}\), which means that there was no noticeable thread imbalance.

• **PCSR ≈ P_{CMP}** Performance did not improve by removing the memory references that are introduced due to sparsity. This implies that the sparse kernel has a similar performance to an equivalent dense kernel.

• **PCSR < P_{CML} and/or P_{CSR} < P_{IMB}** In this case SpMV suffers from excessive cache misses and/or thread imbalance, so it can benefit from an optimization that improves locality for accesses to \(x\) and/or redistributes the workload among threads either statically or dynamically.
• $P_{CMP} < P_{MB}$ This suggests that SpMV using the CSR format might be limited by computational bottlenecks, based on the following analysis:

$$P_{CMP} < P_{MB} \Rightarrow \frac{P_{CMP}}{P_{MB}} < 1 \Rightarrow \frac{2 \cdot NNZ}{M} < 1 \Rightarrow \frac{B}{2 \cdot NNZ} < \frac{M_{CSR}}{M} \frac{B}{B_{ax}} < 1$$

Since the memory footprint of the matrix in the computation of $P_{src}$ is smaller (we no longer use the colind array), and we have eliminated irregularity as well, we can assume that:

$$M < M_{CSR}$$

Since $B \leq B_{max}$, in order for inequality 1 to hold:

$$B < B_{max}$$

so the kernel is not limited by memory bandwidth. By removing irregularity we also exclude the kernel being limited by excessive cache misses. So there are two possibilities to be considered: either the kernel is imbalanced or suffer from computational bottlenecks. If $P_{CSR} \approx P_{IMB}$ and $P_{CSR} < P_{CMP}$, then it is compute-limited. This condition holds only for matrices that have very short rows, and, as mentioned in Section 2, in this case SpMV using CSR suffers from loop overheads.

Our analysis clearly exposes a diversity in performance bottlenecks on the Intel Xeon Phi manycore architecture. A lot of matrices seem to suffer from thread imbalance, e.g., rajat29, TSOFP_FS_b300_c3, degme, etc. and many from excessive cache misses, e.g., language, CO, parabolic_fem, etc. Also note that some matrices seem to have competing bottlenecks, e.g., helm2d03, Hamrle3, thermal2, etc. This diversity justifies a SpMV optimizer that is input-adaptive. Towards this direction, and based on the observations made during this analysis, in the
following section we are going to design our first matrix classification algorithm, that is going to form the decision-making component of our adaptive optimization approach.

C. Profiling-Based Classifier

Our first approach to solving the classification problem involves a hand-crafted classification algorithm that relies on the performance bounds defined in Section 3-B to determine the class of a matrix. In order to estimate the performance bounds, we collect data during an online profiling phase. Thus, we will henceforth refer to this classifier as the profiling-based classifier.

The algorithm is depicted in Figure 4. The classification is performed by comparisons between the class-specific performance bounds and by evaluating their impact on the baseline CSR kernel. The values of \( T_{CMP} \), \( T_{CML} \), and \( T_{IMB} \) have been empirically set, taking into account how naive or realistic each bound is and by verifying whether performance actually improves by applying an optimization assigned to the selected class. Essentially, these thresholds determine whether it is worth optimizing for the corresponding issue. We must note that this classifier implicitly deduces architectural characteristics of the underlying platform, as it relies on profiling data, making it architecture-agnostic.

```plaintext
1: procedure CLASSIFY\( (P_{CSR}, P_{MB}, P_{CML}, P_{IMB}, P_{CMP}, P_{PEAK}) \)
2: if \( (P_{MB} \approx P_{CML} \approx P_{IMB}) \) then
3:     \( \text{class} \leftarrow MB \)
4: end if
5: if \( (P_{CMP} > P_{PEAK} \text{ and } P_{CML} > T_{CMP}) \) or
6:     \( (P_{CMP} < P_{MB} \text{ and } P_{PEAK} > T_{CMP}) \) then
7:     \( \text{class} \leftarrow CML \)
8: end if
9: if \( (P_{IMB} > P_{CML} \text{ and } P_{PEAK} > T_{IMB}) \) then
10: \( \text{class} \leftarrow IMB \)
11: end if
12: if \( (P_{CML} > P_{IMB} \text{ and } P_{PEAK} > T_{CML}) \) then
13: \( \text{class} \leftarrow CML \)
14: end if
15: return class
```

Figure 4 Profiling-based classifier

D. Feature-Based Classifier

We also leverage machine learning to build a classifier. The advantage of this approach is that, given a set of classes, the classification rules can be automatically deduced based on a training data set. The classifier is trained on a predefined set of matrices during an offline stage, and only requires a small number of structural features to be extracted from the input matrix on-the-fly. We will refer to this classifier as the feature-based classifier.

We experiment with a Decision Tree classifier and a Naive Bayes classifier and employ supervised learning algorithms to generate them. Decision Tree learning is performed using an optimized version of the CART algorithm, which has a runtime cost of \( O(N_{\text{features}} N_{\text{samples}} \log N_{\text{samples}}) \) for the construction of the tree, where \( N_{\text{features}} \) is the number of features used and \( N_{\text{samples}} \) is the number of samples used to build the classifier. The Naive Bayes classifier is created using Maximum A Posteriori (MAP) training with a Gaussian distribution assumption, which has a time complexity of \( O(N_{\text{features}} N_{\text{samples}}) \). The query time for these classifiers is \( O(\log N_{\text{samples}}) \) and \( O(N_{\text{classes}} N_{\text{features}}) \) respectively. We generate both classifiers using the scikit-learn machine learning toolkit.

1) Feature Extraction: This classifier uses real-valued features to perform the classification. Table 1 shows all the features we experimented with, along with the time complexity of their extraction from the input matrix. \( N \) denotes the number of rows in the matrix, \( M \) the number of columns, and \( NNZ \) the number of nonzero elements. \( nnz_i \) is the number of nonzero elements of row \( i \), \( bw_i \) the column distance between the first and last nonzero element of row \( i \), \( \text{dispersion}_i = \frac{nnz_i}{bw_i} \), \( \text{clustering}_i = \frac{ngroups_i}{nnz_i} \), where \( ngroups_i \) is the number of groups formed by consecutive elements in row \( i \) and, finally, \( \text{misses}_i \) is the number of nonzero elements in row \( i \) that can
generate cache misses. We naively say that an element will generate a cache miss when its distance from the previous element in the same row exceeds the cache line size of the system.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Definition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>0: exceeds or 1: fits in LLC</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Density</td>
<td>( \frac{\text{NNZ}}{N \cdot M} )</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>( \text{nnz}_{\text{min}} )</td>
<td>( \min { \text{nnz}_1, \ldots, \text{nnz}_N } )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{nnz}_{\text{max}} )</td>
<td>( \max { \text{nnz}_1, \ldots, \text{nnz}_N } )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{nnz}_{\text{avg}} )</td>
<td>( \frac{1}{N} \sum_{i=1}^{N} \text{nnz}_i )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{nnz}_{\text{sd}} )</td>
<td>( \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\text{nnz}<em>i - \text{nnz}</em>{\text{avg}})^2} )</td>
<td>Θ(2N)</td>
</tr>
<tr>
<td>( \text{bw}_{\text{min}} )</td>
<td>( \min { \text{bw}_1, \ldots, \text{bw}_N } )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{bw}_{\text{max}} )</td>
<td>( \max { \text{bw}_1, \ldots, \text{bw}_N } )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{bw}_{\text{avg}} )</td>
<td>( \frac{1}{N} \sum_{i=1}^{N} \text{bw}_i )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{bw}_{\text{sd}} )</td>
<td>( \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\text{bw}<em>i - \text{bw}</em>{\text{avg}})^2} )</td>
<td>Θ(2N)</td>
</tr>
<tr>
<td>( \text{dispersion}_{\text{avg}} )</td>
<td>( \frac{1}{N} \sum_{i=1}^{N} \text{dispersion}_i )</td>
<td>Θ(N)</td>
</tr>
<tr>
<td>( \text{dispersion}_{\text{sd}} )</td>
<td>( \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\text{dispersion}<em>i - \text{dispersion}</em>{\text{avg}})^2} )</td>
<td>Θ(2N)</td>
</tr>
<tr>
<td>clustering</td>
<td>( \frac{1}{N} \sum_{i=1}^{N} \text{clustering}_i )</td>
<td>Θ(NNZ)</td>
</tr>
<tr>
<td>miss_ratio</td>
<td>( \frac{1}{N} \sum_{i=1}^{N} \text{misses}_i )</td>
<td>Θ(NNZ)</td>
</tr>
</tbody>
</table>

Table 1 Sparse matrix features used for classification

Most features are parameters that represent comprehensive characteristics of a sparse matrix and are closely related to SpMV performance. For example, the number of nonzero elements per row can reveal workload imbalance during SpMV execution. If a matrix contains very uneven row lengths and a row-partitioning scheme is being employed for workload distribution, then threads that are assigned long rows will take longer to execute resulting in thread imbalance. Uneven row lengths can easily be detected through the corresponding metrics in Table 1 (see \( \text{nnz}_{\text{min, max, avg, sd}} \)). Similarly, irregularity in the accesses to the input vector, which can lead to excessive cache misses, can be estimated by examining how spread out the elements in every row are (see \( \text{dispersion}_{\text{avg, sd}} \)).

2) Training Data Selection: We train our classifiers with a data set consisting of sparse matrices from the University of Florida Sparse Matrix Collection [21]. We have selected a matrix suite consisting of 125 matrices
from a wide variety of application domains, to avoid being biased towards a specific sparsity pattern. We must note here that we are examining all matrix sizes, not only matrices exceeding the system’s LLC size.

3) Training Data Labeling: Labeling refers to the process of assigning a class to each matrix that will be used in the supervised training process of our classifier. Since the class of a matrix cannot be determined in a straightforward manner, we use our profiling-based classifier for this purpose. An issue that arises by this choice, is that the validity of the labels and consequently, the accuracy of the trained classifier, depends on the behavior of the profiling-based classifier. Since the decision rules in the profiling-based classifier have been empirically set, a matrix can be mislabeled if it falls on the boundaries that separate classes.

E. Optimization Pool

A multitude of optimization techniques and specialized formats have been proposed in the literature for improving the performance of SpMV. Most require a preprocessing phase, either to modify the sparsity pattern of the matrix itself or to generate new data structures for a representation of the matrix that results in higher compression ratios of its memory footprint, better load balancing of the workload among threads, etc. This preprocessing overhead needs to remain low, as it may offset the benefit of applying the optimization, in particular when only tens to a couple hundred iterations are required for a numerical solver to converge.

As most legacy code in scientific applications uses the general-purpose CSR format for sparse computations, we decided, for the sake of simplicity and in order to minimize the preprocessing costs, to incorporate only CSR-based optimizations. We currently experiment with a single optimization per class, with the exception of the IMB class, where a second level of optimization-selection is performed based on structural features of the matrix. However, our methodology is not tied to specific optimizations and can be easily extended to incorporate any optimization proposed in the literature, as long as it has been shown to be effective for the corresponding class.

Table 2 gives a mapping of classes to optimizations. For matrices that are memory bandwidth-limited, we employ a rather simple compression scheme, just for illustrative purposes. We use delta indexing on the column indices of the nonzero elements of the matrix, a technique which was originally applied to SpMV by Pooch and Nieder [22]. We use 8- or 16-bit deltas wherever possible, but never both, in order to limit the branching overhead during SpMV computation. For the CML class of matrices, we employ one of the most important techniques for tolerating increasing cache miss latencies, e.g., prefetching. Since the source of excessive cache misses in SpMV is a result of indirect memory addressing on the input vector, which cannot be efficiently tackled by current hardware prefetching mechanisms, software prefetching was applied. A single prefetch instruction was inserted in the inner loop of SpMV, with a fixed prefetch distance (although it can be tuned on a matrix basis for a higher performance gain). The data is prefetched in the L1 cache. For matrices that suffer from thread imbalance, we include multiple optimizations, as thread imbalance can be a result of either workload imbalance (number of nonzero elements per thread) or imbalance in the number of cache misses generated by each thread or computational unevenness. We select the final optimization based on structural features of the matrix. The first subcategory includes matrices with highly uneven row lengths that we detect by comparing the $\text{nnz}_{\text{max}}$ and $\text{nnz}_{\text{avg}}$ features from Table 2. For these matrices, we basically decompose the matrix into two parts stored in a modified CSR format, depicted in Figure 5. SpMV is performed in two steps, described in Figure 6. First, SpMV is performed as usual by skipping the long rows and, then long rows are computed with a different assignment of computation to threads. Every row is computed by all threads and a reduction of partial results follows. For the second subcategory, which we detect with the bw $\text{sd}$ feature, we employ the auto scheduling policy available in the OpenMP runtime system [23]. When the auto schedule is specified, the decision regarding scheduling is delegated to the compiler, which has the freedom to choose any possible mapping of iterations (rows in this case) to threads. For the rest, we use the dynamic scheduling policy of OpenMP. Finally, for compute-limited matrices, we combine inner loop unrolling with vectorization.

<table>
<thead>
<tr>
<th>Class</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB</td>
<td>column index compression through delta encoding [22]</td>
</tr>
<tr>
<td>CML</td>
<td>software prefetching on vector x</td>
</tr>
<tr>
<td>IMB</td>
<td>matrix decomposition, auto or dynamic scheduling (OpenMP)</td>
</tr>
<tr>
<td>CMP</td>
<td>inner loop unrolling + vectorization</td>
</tr>
</tbody>
</table>

Table 2 Mapping of matrix classes to optimizations
4. Experimental Evaluation

Our experimental evaluation focuses on two aspects: matrix classification accuracy and the overall benefit of applying our methodology for optimizing SpMV.

A. Experimental Setup and Methodology

Our experiments are performed on the Intel Xeon Phi co-processor, codename Knights Corner (KNC). Table 3 lists the technical specifications of the test platform in more detail. For the compilation of the software involved in our performance evaluation, we use Intel ICC-15.0.0, as well as the OpenMP parallel programming API. In all cases, we use 56 cores with full SMT (total 224 threads), and enforce the compact thread affinity policy. To simulate the typical SpMV computation in scientific applications, we used 64-bit, double precision floating point values for the nonzero elements.

<table>
<thead>
<tr>
<th>Codename</th>
<th>Knights Corner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Intel Xeon Phi 3120P</td>
</tr>
<tr>
<td>Microarchitecture</td>
<td>Intel Many Integrated Core</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>1.10 GHz</td>
</tr>
<tr>
<td>L1 cache (D/I)</td>
<td>32 KiB/32 KiB</td>
</tr>
</tbody>
</table>
Table 3 Technical characteristics of Intel Xeon Phi

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 cache</td>
<td>512 KiB (per core)</td>
</tr>
<tr>
<td>Cores/Threads</td>
<td>57/228</td>
</tr>
<tr>
<td>STREAM b/w [20]</td>
<td>119 GB/s</td>
</tr>
</tbody>
</table>

Table 4 Feature-based classifiers for Intel Xeon Phi

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Features</th>
<th>Complexity</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>profiling-based</td>
<td>size, bw[avg, sd], nnz[min, max, avg, sd], miss ratio, dispersion✓</td>
<td>O(NNZ)</td>
<td>77</td>
</tr>
<tr>
<td>feature-based</td>
<td>nnz[min, max, sd], bw[avg], dispersion[avg, sd]</td>
<td>O(N)</td>
<td>75</td>
</tr>
</tbody>
</table>

Table 4 Feature-based classifiers for Intel Xeon Phi

B. Classifier Accuracy

First, we evaluate our feature-based classifier in terms of accuracy, assuming the labels generated by the profiling-based classifier are correct. The profiling-based classifier is not eligible to such an analysis, as we currently have no way of accurately deciding upon the major performance bottleneck of a matrix and, thus, correctly evaluating this classifier. Using optimizations to determine the label of a matrix would require more than one optimization per class in order to be safe, so we do not currently follow this approach.

We estimate how accurately our models perform in practice using Leave-One-Out cross validation. According to this methodology, for a training set of k matrices (125 in our case), k experiments are performed. For each experiment k − 1 matrices are used for training and one for testing. The reported accuracy is the fraction of k matrices for which the classifier correctly predicts their class, i.e. the prediction matches the original labeling of the matrix. This metric is important, as it determines the suitability of the selected optimization for the target matrix. However, it is not decisive, contrary to standard classification problems, in the sense that it depends on the correct labeling of each matrix, which is not straightforward in some cases. There exist matrices that have competing performance bottlenecks, and, thus, their labeling is ambiguous. In fact, this allows us to tolerate a less accurate labeling algorithm, such as our profiling-based classifier. It also means that our classifier might improve the performance of SpMV even in the case of a misprediction. Thus, achieving the highest possible accuracy is desirable, but not of the highest priority.

We note that the feature-based classifier with the highest accuracy reported in Table 4 is used.

C. Performance of selected optimizations

Contrary to standard classification problems, our key measure of success is not classifier accuracy - rather, we aim to maximize the average performance improvement of optimization selection over two baseline implementations, as well as the Intel MKL library. In this way, we designate the value of optimization selection for SpMV. Our first baseline is the CSR SpMV kernel with no vectorization or prefetching applied, while the second is the default CSR kernel compiled with the -O3 flag. In both baseline implementations, the matrix is statically partitioned into row blocks so that each block has approximately equal number of nonzero elements, and each block is assigned to a single thread. This is the typical partitioning scheme for SpMV.

Figure 7 presents raw SpMV performance achieved by MKL using the CSR format, our baseline CSR implementations, the optimizations selected by our classifiers, and the best optimization among our pool of optimizations. The class of each matrix according to the profiling-based classifier is also provided. We note that the feature-based classifier with the highest accuracy reported in Table 4 is used.

Our initial observation concerns the diversity of performance bottlenecks detected by our classifier, depicted by the labeling of the matrices. This was expected, based on our bound and bottleneck analysis in Section III-B, and is mainly due to the architectural characteristics of the co-processor, which result in a quite balanced representation of each class. The co-processor has a large amount of cores/threads, which favor workload imbalance, and a very expensive (an order of magnitude larger compared to multicores) cache miss latency, which further exposes irregularity in the accesses to the input vector. Thus, there are many matrices that fall out of the standard class of memory bandwidth-bound matrices. Our classifiers manage to successfully capture this trend and select proper optimizations due to their input- and architecture-awareness. Taking a closer look at
Figure 7, we notice that our classifiers do not match the best optimization in a few cases, indicating that they might be misclassifying matrices. Actually, this occurs mostly with matrices that have competing bottlenecks, e.g., Chebyshev4, Hamrle3, G3 circuit, etc. Nonetheless, in some cases, the feature-based classifier manages to select a better optimization, even though it has been trained with labels generated by the profiling-based classifier. In total, our profiling-based classifier achieves a 2.2× speedup over MKL, 1.96× over -O3 and 1.41× over our baseline, while our feature-based classifier 2.15×, 1.93× and 1.37× respectively. We estimate that the performance gap between MKL and our baselines is due to a less efficient partitioning of matrix rows to threads. Our results also demonstrate that blindly applying the -O3 optimization level for SpMV leads to inferior results for the majority of matrices compared to a bare-metal implementation. The -O3 flag applies vectorization to the CSR kernel, but in an inefficient for the architecture way, as it does not use the gather instruction available on Xeon Phi. Furthermore, it adds prefetching instructions for the colind and values arrays, which is not beneficial for the majority of matrices, as it generates more memory traffic.

Figure 7 Performance landscape on Intel Xeon Phi. MKL refers to CSR SpMV using the Intel MKL library, baseline refers to CSR SpMV with no vectorization and prefetching, feat refers to the feature-based classifier, prof to the profiling-based classifier, and -O3 to CSR SpMV compiled with -O3.

We further emphasize the need for an adaptive optimizer for SpMV by illustrating the performance slowdown in case of “ill-suited” optimizations. Table 5 shows the percentage of times both our classifiers, as well as a random classifier, achieve the best performance for a matrix (taking into account baseline performance as well, i.e., no optimization). For those cases where the best optimization is not selected, we provide the average and max performance slowdown. For the random classifier we exclude the matrix decomposition optimization, as it is a custom optimization that would never be selected for general-purpose SpMV optimization. Results indicate that randomly picking optimizations for SpMV can severely hurt performance, with an average slowdown of up to 1.69×.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Predicts Best Opt (%)</th>
<th>Average Slowdown</th>
<th>Max Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>profiling-based</td>
<td>86</td>
<td>1.11x</td>
<td>1.45x</td>
</tr>
<tr>
<td>feature-based</td>
<td>76</td>
<td>1.17x</td>
<td>2.19x</td>
</tr>
<tr>
<td>random</td>
<td>25</td>
<td>1.43 - 1.69x</td>
<td>3.39 - 8.69x</td>
</tr>
</tbody>
</table>

Table 5 Classifier performance statistics
D. Optimization overhead

Non-preconditioned iterative methods for the solution of linear systems, including variations of the Conjugate Gradient (CG) and the Generalized Minimal Residual (GMRES) methods, usually require hundreds to thousands of iterations to converge, making it easier to amortize the preprocessing costs of the optimization process, e.g., optimization selection, format conversion, runtime code generation, etc. However, it is quite common in real-life applications to use preconditioned versions of these methods to accelerate convergence [24]. In this case, the number of iterations is much smaller, ranging from dozens to hundreds, thus, limiting the online overhead that can be tolerated. Specifically, if $T_{\text{solver}}$ and $T'_{\text{solver}}$ are the total execution times of the original and optimized solver respectively, and $T_{\text{pre}}$ the preprocessing cost that needs to be amortized, then:

$$T_{\text{solver}} + T_{\text{pre}} < T_{\text{solver}}$$
$$N_{\text{iters}} \cdot T'_{\text{iter}} + T_{\text{pre}} < N_{\text{iters}} \cdot T_{\text{iter}}$$
$$N_{\text{iters}} \cdot (T'_{\text{spmv}} + T_{\text{other}}) + T_{\text{pre}} < N_{\text{iters}} \cdot (T_{\text{spmv}} + T_{\text{other}})$$
$$N_{\text{iters}} \cdot T'_{\text{spmv}} + T_{\text{pre}} < N_{\text{iters}} \cdot T_{\text{spmv}}$$
$$N_{\text{iters}} > T_{\text{pre}} / (T_{\text{spmv}} - T'_{\text{spmv}}) \quad (1)$$

where $N_{\text{iters}}$ is the number of iterations required for the solver to converge, and $T_{\text{spmv}}$ and $T'_{\text{spmv}}$ the execution time of SpMV before and after optimization.

Our profiling-based optimization selection approach comprises two steps: running a number of micro-benchmarks on the input matrix and then applying our empirically-tuned classification algorithm to select the appropriate optimization. On the other hand, the online stage of the feature-based optimization selection methodology comes down to extracting the required features from the input matrix and then predicting its class using the pre-trained classifier.

The overall runtime overhead of both approaches, including the application of the optimization, was measured in SpMV operations, i.e., we report the ratio $T_{\text{pre}} / T_{\text{spmv}}$. We also measure the overhead of a trivial optimizer, that simply runs one optimization per class and selects the best for each matrix. Table 6 shows the average overhead of each classifier in MKL SpMV operations, the associated average performance loss over the trivial optimizer, and the minimum number of iterations required for the optimizer to be beneficial in the context of an iterative solver according to Equation 1. As expected, the feature-based classifier has a significant advantage over its alternatives, with a small sacrifice in performance.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Overhead</th>
<th>Performance loss (%)</th>
<th>Solver Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>trivial</td>
<td>49</td>
<td>0</td>
<td>86</td>
</tr>
<tr>
<td>profiling-based</td>
<td>20</td>
<td>-2.3</td>
<td>36</td>
</tr>
<tr>
<td>feature-based</td>
<td>10</td>
<td>-4.5</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 6 Optimizer overhead vs performance trade-off

E. Discussion

Our evaluation has demonstrated the benefit of following a matrix-adaptive optimization selection approach for the SpMV kernel, by achieving significant speedups over popular baseline implementations in a large and diverse test suite. In our profiling-based classifier, adaptivity comes through the performance bounds used to classify the matrix, while in our feature-based classifier, both the structural features and the use of the profiling-based classifier during training contribute. Most importantly, this adaptivity comes at a low cost, which is a requirement for problems that require a small number of iterations to converge, e.g., preconditioned solvers. However, for problems that can amortize higher preprocessing costs, better performance can be attained by incorporating more sophisticated optimizations and exploring combined optimizations for matrices that have competing bottlenecks. By design, our classifiers can detect competing bottlenecks, but actually taking advantage of this information would increase complexity and is left for future work.

5. Related Work

Different sparse matrices have different sparsity patterns, and different architectures have different strengths and weaknesses. In order to achieve the best SpMV performance for the target sparse matrix on the target platform, an autotuning approach has long been considered to be beneficial. The first autotuning approaches attempted to
tune parameters of specific sparse matrix storage formats. Towards this direction, the Optimized Sparse Kernel Interface (OSKI) library [25] was developed as a collection of high performance sparse matrix operation primitives on single core processors. It relies on the SPARSITY framework [26] to tune the SpMV kernel, by applying multiple optimizations, including register blocking and cache blocking. Autotuning has also been used to find the best block and slice sizes of the input sparse matrix on modern CMPs and GPUs [27].

There have been some research efforts closer to our work. The cSpMV framework [28] is the first framework that analyzes the input sparse matrix at runtime, and recommends the best representation of the given sparse matrix, but it is restricted to GPU platforms. Towards the same direction, the authors in [29] present an analytical and profile-based performance modeling to predict the execution time of SpMV on GPUs using different sparse matrix storage formats, in order to select the most efficient format for the target matrix. For each format under consideration, they establish a relationship between the number of nonzero elements per row in the matrix and the execution time of SpMV using that format, thus encapsulating to some degree the structure of the matrix in their methodology. Similarly, in [30], the authors propose a probabilistic model to estimate the execution time of SpMV on GPUs for different sparse matrix formats. They define a probability mass function to analyze the sparsity pattern of the target matrix and use it to estimate the compression efficiency of every format they examine. Combined with the hardware parameters of the GPU, they predict the performance of SpMV for every format. Since compression efficiency is the determinant factor in this approach, it is mainly targeted for memory bandwidth-bound matrices. Closer to our approach is the SMAT autotuning framework [31]. This framework selects the most efficient format for the target matrix using feature parameters of the sparse matrix. It treats the format selection process as a classification problem, with each format under consideration representing a class, and leverages a data mining approach to generate a decision tree to perform the classification, based on the extracted feature parameters of the matrix. The distinguishing advantage of our optimization selection methodology over the aforementioned approaches is that it decouples the decision making from specific optimizations, by predicting the major performance bottleneck of SpMV instead of SpMV execution time using a specific optimization. Thus, in contrary to the above frameworks, where incorporating a new optimization requires either retraining a model or defining a new one, our decision-making approach allows an autotuning framework to be easily extended, simply by assigning the new optimization to one of the classes.

Recent efforts target specifically the Intel Xeon Phi co-processor [32], [33]. In [32], the authors also perform a performance bounds analysis on the SpMV kernel and expose low SIMD efficiency, cache miss latency, and load imbalance as important bottlenecks. They address those issues with a single format, by incrementally applying optimizations that target the aforementioned bottlenecks, and achieve significant speedups. However, their evaluation is performed on a small matrix suite, and the incremental optimizations are shown only when they increase performance. Their results suggest that the format could benefit from an input-adaptive autotuner. In [33], the authors focus on scale-free matrices and design a format that vastly improves performance over Intel MKL. Their work is complementary to ours and could be incorporated as a more sophisticated optimization for irregular matrices.

6. Conclusions

In this paper, we demonstrated that an input-adaptive optimizer can provide significant speedups for the SpMV kernel on a manycore processor. In the proposed optimization method, we formulate optimization selection as a classification problem, with each class corresponding to a performance bottleneck. We classify the input matrix based on its prevailing performance bottleneck and then apply a targeted optimization to tackle it. We describe two classifiers: a) a profiling-based classifier that relies on performance bounds to perform the decision making, and b) a trained classifier that relies on comprehensive structural features. All aspects of the optimization process focus on minimizing online overheads in order to be applicable in real-life scenarios. Experimental evaluation of 125 sparse matrices on the Intel Xeon Phi co-processor has demonstrated that our methodology is very promising, achieving a 2.20× average speedup over the Intel MKL library using the feature-based classifier, with an online overhead that is approximately 5× smaller than that of a trivial optimizer.

References
Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838.