Applicability of Pattern-based Sparse Matrix Representation for Real Applications

Mehmet Belgin¹, Godmar Back, Calvin J. Ribbens

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 24061

Abstract

Pattern-based representation (PBR) is a novel sparse matrix representation that reduces the index overhead for many matrices without zero-filling and without requiring the identification of dense matrix blocks. The PBR analyzer identifies recurring block nonzero patterns, represents the submatrix consisting of all blocks of this pattern in block coordinate format, and generates custom matrix-vector multiplication kernels for that submatrix. In this way, PBR expresses matrix structure in terms of specialized inner loops, thereby creating locality for repeating structure via the instruction cache, and reducing the amount of index data that must be fetched from memory. In this paper we evaluate the applicability of PBR by testing it on a large set of matrices from the University of Florida sparse matrix collection. We analyze PBR’s suitability for a wide range of problems and identify underlying problem and matrix characteristics that suggest good performance with PBR. We find that PBR is especially promising for problems with underlying 2D/3D geometry.

Keywords: sparse, matrix-vector multiplication, sparse matrix representation, code generation

1. Introduction

Sparse matrix vector multiplication (SMVM) is a dominant computational kernel in many algorithms for solving important computational science problems, including iterative methods for solving linear systems of equations and eigenvalue problems. Unfortunately, SMVM kernels typically achieve only a small fraction of peak performance on modern systems, primarily due to the low ratio of floating point operations to memory accesses in SMVM. A variety of sparse matrix representations are used to reduce the total amount of data that must be retrieved from memory and to avoid arithmetic operations on zeros, but the fact remains that SMVM is memory-bound. For example, if a sparse matrix is represented in the compressed sparse row (CSR) format [1], each floating point multiplication requires two memory accesses that trigger compulsory cache misses: one to retrieve the matrix nonzero element and a second to retrieve its column index. Hence, since accesses to main memory are typically much more expensive than arithmetic operations, SMVM achieves only a small fraction of peak performance.
Recent efforts to improve SMVM performance have taken several approaches. Various blocking methods (e.g., [2, 3, 4, 5, 6, 7]) attempt to reduce memory bandwidth usage in case a matrix contains dense substructures. Instead of recording one index per matrix element, blocked representations record one index per block. However, blocking may require zero-filling, which introduces unnecessary memory and floating point operations, and it cannot be applied if a matrix does not contain dense substructures or if those structures cannot be identified. Reducing the index overhead is also possible by using a run-length encoding if a row contains contiguous columns of nonzeros [8], or in some cases by compressing index arrays or nonzero arrays [9, 10]. Register and cache blocking methods can also reduce memory bandwidth usage [11, 12], but their primary objectives are to improve register reuse and cache locality. Register and cache blocking techniques are combined in the OSKI library [3]. Other recent efforts to understand and improve SMVM performance include [13], [14] and [15].

In [16] we introduced a novel sparse matrix representation, pattern-based representation (PBR), which reduces the indexing overhead for many matrices without zero-filling and without requiring the existence or identification of dense substructures. Instead, PBR exploits a simple analysis that identifies recurring blocks that share the same nonzero pattern. The PBR code generator emits optimized custom codes for each sufficiently recurring block pattern with three or more nonzeros, and keeps only a single pair of indices per block, but it does not keep any indices for nonzeros inside the pattern. Instead, PBR expresses pattern structures as specialized inner loops in custom codes, thereby creating locality for repeating structure via the instruction cache, and reducing the amount of index data that must be fetched from memory. Furthermore, since the nonzero structure of each submatrix is known in detail at compile time, we can make processor-aware choices regarding optimizations such as prefetching and vectorization. A remainder matrix holds nonzeros not contained in frequently recurring patterns, and is represented using CSR.

We presented the idea of PBR in [16], along with an initial performance evaluation based on a standard set of 53 matrices. For a majority of these matrices, we found that a high percentage of nonzeros are covered by PBR, with coverage close to 100% in some cases. We found that PBR can shorten time to solution by up to 3.4×, with 1.4× on average, when compared to CSR in a sequential implementation, and it can also improve time to solution when compared to the OSKI library [3]. However, this evaluation assumed off-line matrix analysis, structure conversion, and code generation/compilation, which prevented a quantitative analysis of overhead vs. benefit. In a more recent paper [17], we described a library that performs all required steps at run time. We also verified that the costs of the analysis and structure conversion steps are $O(N + NNZ)$ for a matrix of dimension $N$ with $NNZ$ nonzeros. In addition, we proposed a model for PBR-SMVM performance and used this model to derive a highly accurate predictor for choosing the best block size. Finally, we showed that PBR matrix analysis costs can be compensated for within a few hundreds of iterations in most cases, but code compilation costs (which can be largely avoided by code caching) might add to this break-even point thousands of additional iterations.

The goal of this paper is to evaluate the applicability of PBR by testing it against a much wider set of problems than in previous work, characterize the types of problems for which PBR is most and least suitable, and demonstrate a real code cache implementation to significantly mitigate code compilation costs. Since PBR’s performance is dependent on matrix nonzero structure, and since matrix structure in turn is dependent on the underlying problem, it is critical to evaluate PBR on problems drawn from as wide a range of application areas as possible. Furthermore, a better understanding of the link between PBR-SMVM performance and underlying matrix structure will guide practitioners in deciding whether or not to use PBR, and may suggest alternatives for algorithm designers who can influence the structure of matrices, e.g., via discretization techniques or matrix reorderings. As a test set we use all real square matrices of dimension 10,000 or larger from the University of Florida sparse matrix collection [18]—a total of 710 matrices drawn from a wide variety of real applications. Since the primary advantage of PBR is in reducing memory bandwidth usage (which is directly related to matrix nonzero coverage), and since the most expensive overhead of PBR is code generation and compilation, we consider these two factors in our experiments.

The remainder of the paper is organized as follows: Section 2 provides an overview of PBR and its implementation, Section 3 describes our experimental results, and Section 4 summarizes and concludes.

### 2. Pattern Based Representation

PBR computes a vector $y$ as the product of a sparse matrix $A$ and a vector $x$: $y = Ax$ by using memory bandwidth more efficiently compared to other techniques, such as the widely used CSR format. In CSR, the matrix nonzeros are stored contiguously in an array $aa$. Two indices record the structure of the matrix: an index $ja[j]$ records the column...
index of the \(j^{th}\) nonzero element, and a row pointer \(i_a[j]\) records at which matrix element row \(i\) begins. Hence, each pair of multiply-add operations is accompanied by at least two memory accesses: one to load the element \(a_a[j]\), and a second to retrieve the column index \(j_a[j]\). Neither of these values is reused within the same invocation of the kernel. Therefore, even if there is maximum reuse of \(x\), performance is limited by the speed with which \(a_a\) and \(j_a\) can be fetched from memory. Except when SMVM is repeatedly called for small matrices that fit in the cache, each of these accesses will encounter compulsory or capacity cache misses.

2.1. Exploiting Recurring Patterns

Because we cannot reduce the number of nonzero values \((a_a)\) that must be read from main memory, our approach focuses on reducing the size of the index data structures. Based on our observation that many matrices can be decomposed into blocks that share identical patterns, we generate customized code for each sufficiently recurring block pattern in the matrix. Since coordinates of each block can be expressed by a single pair of indices, rather than a pair for each nonzero within a block, the matrix can be represented using fewer indices and memory bandwidth usage is reduced. The microstructure of each block is expressed in the machine code of the inner loop that iterates over all blocks of identical structure within a matrix.

We use a simple analysis to identify repeating patterns. Given a block size \(R \times C\), we divide a \(m \times n\) matrix into a grid of \(\left\lfloor \frac{m}{R} \right\rfloor \times \left\lfloor \frac{n}{C} \right\rfloor\) rectangular blocks and count how often each of the possible \(2^{R \times C}\) patterns occurs. We represent the aggregate submatrix for each block pattern by recording block coordinates in BCOO format, along with a “block code,” which is a bit vector of size \(R \times C\) that encodes the nonzero micro-pattern. We exclude two types of blocks. First, we exclude blocks with patterns that include less than three nonzeros, because such patterns would yield little or no reduction in index overhead. Second, we exclude blocks whose patterns do not occur frequently enough to cover a significant number of nonzero elements, because the overhead of dispatching to the kernel specific to their block code may not be amortized. We empirically set this threshold as one thousand. Our analysis aggregates nonzeros that belong to excluded blocks in a remainder matrix, which is stored using conventional CSR representation.

Figure 1 shows an example \(12 \times 12\) matrix in which three recurring patterns occur when using a block size of \(4 \times 4\). Seven of the matrix’s nine blocks exhibit recurring patterns with more than two nonzeros; three remainder elements (shown in red) are in blocks with less than three nonzeros. For this example, PBR reduces index overhead by 35% when compared to CSR (from 46 to 30 integers).

Figure 2 shows how the original matrix is split into a sum of submatrices such that nonzeros of blocks sharing an identical pattern are stored consecutively, and remainder nonzeros are collected in the remainder matrix.

2.2. PBR Library

The PBR library implements a conversion process on an input CSR matrix involving: matrix structure analysis to find recurring patterns, block size selection, structure conversion, optimized code generation and compilation (if
necessary), and loading. An analyzed matrix structure can be saved to disk, allowing re-use of the analysis results for structurally identical matrices.

As the first step, the PBR analyzer performs a structure analysis to determine which block patterns occur in the matrix and how often. This step also calculates the number of writes and reads to/from memory, to be used as parameters to the block size selection step that follows. The asymptotic time complexity of the analysis step is linear in the number of nonzeros contained in the matrix.

In order for PBR to be competitive with CSR, we must choose a block size that yields sufficient nonzero coverage relative to the number of nonzeros in the remainder matrix. Nonzero coverage depends on the choice of block size $R \times C$ and on the cutoff criteria. Currently, PBR assumes square block sizes $R = C = 2, 3, 4, \ldots, 8$. Block sizes larger than $8 \times 8$ tend to increase the number of possible patterns, making it less likely that individual patterns meet the cut-off criteria. Our observations show that the best SMVM performance is not always realized by the block size that provides the highest coverage however, because different block sizes also lead to different memory access characteristics for the $x$ and $y$ vectors. In previous work [17], we explored these factors to describe a simple multiple linear regression model, which is used as a performance predictor to choose a block size that yields optimal or near-optimal performance. The model includes three factors: number of memory reads for matrix nonzeros and block indices, number of reads for the $x$ and $y$ vectors, and the number of writes to $y$.

The structure conversion step creates the block indices and arranges the nonzero values in PBR format for the selected block size. We keep the (row, col) block indices and the matrix nonzero values in two one-dimensional contiguous arrays. As depicted in Figure 2, the nonzeros and indices of all blocks that belong to the same pattern are stored in contiguous slices of these arrays, which guarantees spatial locality in the inner loop of each kernel.

The code generation step generates custom C functions for all qualifying patterns, stores them to disk, and invokes the C compiler to create shared .so object files. Each .so file is dynamically linked into the process’s address space. The outer loop of the SMVM routine iterates over the patterns, invoking the corresponding multiplication function for each.

The shared object modules are created on demand and stored in a repository on disk. Since object modules are specific to only the block pattern and size, they can be reused across multiple uses of the PBR library on the same matrix as well as across multiple matrices that contain the same pattern. Our code cache is designed to hold modules for different target architectures and with different optimization options (e.g., SSE) simultaneously, thus allowing it to be shared by multiple machines on a network.

The generated SMVM pattern-specific kernels include three optimizations, which take advantage of the microstructure detail we know about each matrix: (1) explicit software prefetching, which places data directly into the L1 cache and labels cache entries with the correct temporal locality, thus allowing eviction in preference to other data such as elements of the $x$ and $y$ vectors, which may be reused; (2) vectorization, which exploits SSE2/3 SIMD intrinsics; and (3) parallelism, which uses custom thread pool. Details on these optimizations can be found in [17].
### 3. Experimental Results

In order to assess the applicability of PBR for a wide range of application areas, we use matrices from the University of Florida’s sparse matrix collection [18]. We consider all real square matrices of dimension at least 10,000 contained in the collection, which results in 710 matrices drawn from 599 problems. Each matrix in the collection is associated with a problem kind indicating the application area from which it is taken. Table 1 lists the 26 problem kinds included in our study, split into two groups (following the characterization in [18]): problems with no underlying geometry and problems with 2D or 3D geometry.

We ran the PBR analyzer on each of the 710 matrices, with block size $k \times k$, for $k = 2, \ldots, 8$. To predict the best block size, we used the model described in Section 2.2 with parameters acquired on a 2.8GHz 2-socket quadcore Intel Xeon Harpertown 5400 with 12 MB L2 cache and 4GB RAM per socket.

#### 3.1. Nonzero Coverage with PBR

Since a necessary condition for applying PBR is that a high percentage of nonzeros be covered by PBR, we turn first to this metric. At first glance, PBR nonzero coverage seems somewhat uniformly distributed across the 710 matrices. For example, coverage is greater than 50% for 358 of the matrices (50.4%), leaving approximately half of the matrices with coverage less than 50% with PBR, which suggests that PBR will not achieve noticeable speedups on these matrices. However, for 232 matrices (32.7%) we see coverage over 90% with PBR. Matrices with high PBR nonzero coverage are very likely to benefit significantly from speedups due to reduced memory bandwidth requirements. For example, for these 232 matrices the average reduction in the size of the indexing data structures is 77%, resulting in an reduction of 26% in overall bandwidth requirements.

Figure 3 shows the data split into the two groups defined by Davis [18]: matrices with underlying 2D/3D geometry (left) and matrices lacking such underlying geometry (right). The coverage for most matrices with underlying 2D/3D geometry is good, with 65.6% of the matrices over 50% coverage, and 47% above 90% coverage. For matrices with no 2D/3D geometry there is a more uniform distribution, with numerous examples at all coverage levels. We note that 67 of the 111 matrices with less than 10% coverage in Figure 3 (right) come from a single problem, namely the so-called Reuters problem, where average coverage is actually 0%. This problem generates extremely irregular and sparse matrices where nonzeros represent common occurrences of words in news articles. Figure 4 shows the results from four other problem kinds, including one with very high coverage (structural problems) and others with a mix of coverage results (computational fluid dynamics, optimization and simulation problems).

Applying the PBR analyzer to a given matrix is not expensive, so there is no great cost in checking to see if PBR is likely to perform well on a given matrix. However, to give more informed guidance to practitioners and algorithm developers, we take a closer look at the relationship between matrix structure and nonzero coverage by considering

<table>
<thead>
<tr>
<th>Problem Kind</th>
<th># Matrices</th>
<th>Problem Kind</th>
<th># Matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D/3D</td>
<td>35</td>
<td>chemical process simulation</td>
<td>26</td>
</tr>
<tr>
<td>acoustics</td>
<td>5</td>
<td>circuit simulation</td>
<td>49</td>
</tr>
<tr>
<td>computational fluid dynamics</td>
<td>40</td>
<td>combinatorial</td>
<td>5</td>
</tr>
<tr>
<td>computer graphics/vision</td>
<td>2</td>
<td>counter-example</td>
<td>1</td>
</tr>
<tr>
<td>electromagnetics</td>
<td>8</td>
<td>directed graph</td>
<td>11</td>
</tr>
<tr>
<td>materials</td>
<td>25</td>
<td>directed weighted graph</td>
<td>16</td>
</tr>
<tr>
<td>model reduction</td>
<td>45</td>
<td>directed weighted random graph</td>
<td>1</td>
</tr>
<tr>
<td>random 2D/3D</td>
<td>2</td>
<td>economic</td>
<td>44</td>
</tr>
<tr>
<td>semiconductor device</td>
<td>28</td>
<td>frequency-domain circuit simulation</td>
<td>4</td>
</tr>
<tr>
<td>structural</td>
<td>118</td>
<td>optimization</td>
<td>89</td>
</tr>
<tr>
<td>thermal</td>
<td>15</td>
<td>power network</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>statistical/mathematical</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>theoretical/quantum chemistry</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>undirected graph</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>undirected weighted graph</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 1: Problem kinds considered (using terms from [18]) and number of matrices for each.
four extreme cases in Figure 3: less than 10% coverage, greater than 90% coverage, with and without underlying 2D/3D geometry. For the 111 matrices with poor coverage and no underlying 2D/3D geometry, an average of over 90% of the nonzeros end up in the remainder matrix because they are found in blocks with less than three nonzeros. The dominance of 1- and 2-nonzero blocks in these extremely sparse and irregularly structured matrices, which lack recurring relationships between rows or columns, implies that PBR is unlikely to yield noticeable speedups on such problems. On the other hand, for the 22 poorly covered matrices from the group with underlying 2D/3D geometry, more than a quarter of nonzeros (26%) were rejected because they did not cover at least 1000 nonzeros, although the pattern they belong to has 3 or more nonzeros. Hence, there is slightly more hope that the coverage for these problems with underlying geometric structure would improve if larger versions of these problems are solved. In fact, for the 15 CFD problems with coverage less than 40% (Figure 4, upper right), an average of 27% of the nonzeros are in blocks that failed to achieve the required 1000 nonzero total, suggesting that larger versions of these problems would be more amenable to PBR.

Turning to the high (>90%) coverage problems, we see that good coverage stems most often from many occurrences of a very small number of patterns. For problems with underlying 2D/3D geometry this is not surprising, since many such problems correspond to discretized PDEs, where finite difference stencils or finite element formulations yield regularly structured sparse matrices. For example, for the 152 high coverage problems with underlying 2D/3D geometry, the most frequently occurring five patterns (for each matrix) account for an average of 82% of the nonzeros; in fact, 94 of these 152 matrices get more than 90% coverage from only five patterns or less. The dominance of a few patterns is not as strong for problems with no 2D/3D geometry. Only 26 of the 80 matrices in this category achieve 90% coverage with five patterns or less.

3.2. Using a Code Repository

Code generation and compilation of pattern-specific SMVM kernels is costly. If code generation is needed, its overhead adds to the overhead of performing the PBR analysis and structure conversion, thus increasing the number of SMVM operations needed to amortize this overhead. In our experiments [17], we found that the break even point for PBR can range from an average of 459 operations if no code generation is required to an average of 2682 if code for all qualifying patterns must be created (for matrices with more than 50% coverage).

Fortunately, the use of a code repository can reduce or eliminate the cost of code generation in many cases. We have implemented a code repository, which caches compiled kernels as dynamically loadable object modules. In this section, we evaluate the storage requirements and hit rates for this cache for the test set of matrices considered in this
Figure 4: PBR nonzero coverage for four example problem kinds: structural (top left) and computational fluid dynamics (top right) have 2D/3D structure; optimization (bottom left) and circuit simulation (bottom right) do not.

paper. The right half of Figure 5 shows how many qualifying patterns occur, on average, in 10 randomly selected subsets with varying sizes. In this experiment, we choose subsets from only the 358 out of all 710 matrices that achieved at least 50% non-zero coverage, for which PBR is likely to provide speedups. The figure also shows how many of all qualifying patterns are unique.

We draw two conclusions from this experiment. First, the experiment shows that only a small fraction of all theoretically possible patterns qualify—only 15595 for the test set we considered. This result is explained by the use of the cutoff-criteria which eliminates rarely occurring patterns as well as all 1-bit and 2-bit patterns. Consequently, it becomes feasible to store all patterns on disk. For example, the optimized and stripped GCC-compiled object code for Intel’s Harpertown architecture for the patterns occurring in the 358 matrices with more than 50% coverage requires only 60MB of disk space for the block sizes that provide the best coverage. Even if all block sizes and all 710 matrices were considered, the required space would be less than 1 GB.

Second, the figure shows that the number of unique patterns encountered grows much more slowly than the total number of patterns, indicating a high degree of pattern reuse and suggesting that the cache performs well. The left half of the figure quantifies the cache hit rate we observed. The cache is primed with a randomly selected subset of $N$ matrices and the average cache hit rate for all $(358 - N)$ matrices not used for priming the cache is computed. This experiment is repeated for 10 different random subsets; each data point represents the average of these 10 runs.
The results show that 70 matrices are needed to achieve a hit rate of 70%, 140 matrices to achieve a hit rate of 80%, which grows to 86% for 250 matrices. This results suggests that even with a set of matrices as diverse as this one, a code repository will yield substantial savings in kernel generation and compilation. Finally, note that if all matrices are drawn from one problem kind (e.g., on a computational resource used by only one research group), the code repository will likely yield higher hit rates while still being of manageable size.

4. Conclusions

We have explored the applicability of PBR to a range of problem types drawn from a large sparse matrix repository. A necessary condition for applying PBR, namely high matrix nonzero coverage, is most commonly found in matrices generated from problems with underlying 2D or 3D geometry. This is not surprising, since many such problems correspond to discretizations of physical regions, where sets of equations are generated corresponding to local interactions. If the same discretization techniques are applied throughout the problem domain, similar nonzero structure is likely to be repeated throughout the matrix. The advantage of PBR is that this structure can be identified cheaply and automatically, so that discretization and matrix generation can be decoupled from matrix (re-)ordering motivated by performance. On average, we found that PBR is less likely to be helpful for problems which do not have underlying physical geometries. However, there are numerous counterexamples where PBR does achieve excellent coverage. Fortunately, running the PBR analyzer is not expensive, so there is little cost in checking if PBR will achieve high coverage on a given matrix. Furthermore, a code repository can reduce the cost of kernel generation and compilation substantially. For some instances, it is practical to keep virtually all the kernels that are likely to be seen for matrices corresponding to a given problem domain.

In the future, better understanding of underlying characteristics that make problems amenable (or not) to PBR may suggest discretization or matrix reordering schemes that can increase PBR coverage and yield greater performance. In addition, we will incorporate PBR in production codes, to evaluate speedup including the benefits of the code repository.
References