Optimization by Runtime Specialization for Sparse Matrix-Vector Multiplication

Sam Kamin[†] María Jesús Garzarán[†] Barış Aktemur[‡] Danqing Xu[†] Buse Yılmaz[‡] Zhongbo Chen[†]

†: Department of Computer Science
University of Illinois at Urbana-Champaign, USA
{kamin,garzaran,dxu10,chen208}@illinois.edu

‡: Department of Computer Science
Ozyegin University, Turkey
baris.aktemur@ozyegin.edu.tr, buse.yilmaz@ozu.edu.tr

Abstract

Runtime specialization optimizes programs based on partial information available only at run time. It is applicable when some input data is used repeatedly while other input data varies. This technique has the potential of generating highly efficient codes.

In this paper, we explore the potential for obtaining speedups for sparse matrix-dense vector multiplication using runtime specialization, in the case where a single matrix is to be multiplied by many vectors. We experiment with five methods involving runtime specialization, comparing them to methods that do not (including Intel's MKL library). For this work, our focus is the evaluation of the speedups that can be obtained with runtime specialization without considering the overheads of the code generation.

Our experiments use 23 matrices from the Matrix Market and Florida collections, and run on five different machines. In 94 of those 115 cases, the specialized code runs faster than any version without specialization. If we only use specialization, the average speedup with respect to Intel's MKL library ranges from 1.44x to 1.77x, depending on the machine. We have also found that the best method depends on the matrix and machine; no method is best for all matrices and machines.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code Generation

Keywords program specialization, sparse matrix-vector multiplication, performance evaluation

1. Introduction

The technique of *program specialization* begins with the observation that many computations get their inputs in two parts: an early, stable part, and a late, dynamic part. One then asks the question: Given the early data, can we fashion a new, specialized, program that will process the dynamic data very efficiently? For example, in some numerical applications, a single matrix M is multiplied by many vectors v; M is early and stable, the vectors late and dynamic. Can we create a very efficient function $\texttt{multBy}_M(v, w)$ to multiply M by an input vector v (placing the result in w)?

GPCE, '14, September 15-16, 2014, Västerås, Sweden.
Copyright © 2014 ACM 978-1-4503-3161-6/14/09…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Program specialization is a well-studied area [13, 30, 32]. Research has produced many examples of programs, in many problem domains, that have been optimized by specialization. However, most of the work has focused on languages and infrastructure, rather than realistic applications. Take the matrix multiplication example again. The "optimal" approach is simply to unfold the calculation. Instead of a loop iterating over M and v, $\texttt{multBy}_M$ consists of a long sequence of assignment statements of the form

$$\texttt{w[}i\texttt{] += } M_{i,j_0} \texttt{ * v[}j_0\texttt{] + } M_{i,j_1} \texttt{ * v[}j_1\texttt{] + ...;}$$

where the italicized parts — i, M_{i,j_0}, j_0, etc. — are *fixed* values, not variables or subscripted arrays. (The simpler case of vector-vector dot product is a standard "toy" example in this field [16]; a variation of sparse matrix-vector multiplication was recently posed as a Shonan Challenge [8]). This code is "optimal" in the sense of producing the minimum instruction count; a standard Compressed-Sparse-Row (CSR) loop (see Section 2.2) will execute perhaps five times as many instructions as this unfolded code. They will, of course, execute the same number of floating-point operations; the additional instructions are all integer, control, or load operations.

However, it will come as no surprise to those who work in the area of high-performance computing that instruction count tells only a part of the story. Execution speed is affected by such factors as the quality of the code (e.g. register usage), and memory system performance. Traditionally, the latter is concerned primarily with avoiding cache misses when accessing v and w (with accesses to M being purely sequential and therefore not subject to optimization); a new concern that arises here is access to the code itself.

This paper addresses the potential for optimizing sparse matrix–dense vector multiplication by specialization relative to the matrix M, using matrices of realistic size and structure. To that end, we explore a variety of methods and report on their efficiency. The methods (described in detail in Section 2) are these:

Compressed sparse row (CSR). This is the straightforward implementation using the most traditional representation for sparse matrices. Some efficiency is gained by unrolling the inner loop; we refer to CSR with the inner loop unrolled u times as $\texttt{CSR}_u$.

Unfolding. This is the simple unfolded code described above.

CSRbyNZ. This method generates a loop for each group of rows that contain a given number of non-zeros [27]. In effect, this provides a perfect unrolling of the inner loop of CSR.

Stencil. This method analyzes the matrix to find the patterns of non-zero entries in each row of M, and generates, for each pattern, a loop that handles all the rows that have that pattern.

GenOSKI. This method analyzes the matrix to find the patterns of non-zero entries in each block of size $r \times c$, and for each pattern, generates straight-line code [9]. A motivation of this method is to avoid the zero-fill problem of OSKI [21], that generates efficient per-block code by inserting some zeros into the matrix data.

We also have compared our results with 3 state of the art libraries: the Intel MKL library [5], BiCSB [12] and CSX [24]. BiCSB [2] is implemented on top of CSB [11], a new parallel sparse matrix data structure that allows efficient SpMV on multicores. BiCSB requires some restructuring of the data, but no runtime generation of the code. CSX [3] is based on the Compressed Sparse eXtended (CSX) format that allows for a flexible storage format to support a variety of patterns within the sparse matrix, such as horizontal, vertical, diagonal, anti-diagonal, or blocks.

Notice that not all of these methods and libraries involve runtime specialization. We can classify them in three groups: those that are completely generic and operate on the standard CSR representation (CSR); those that require some restructuring of the data but no runtime generation of code (CSB, BiCSB, and OSKI); and those that require runtime code generation (Unfolding, CSRbyNZ, Stencil, GenOSKI[1], and CSX). The distinction matters because it refers to the latency of each method — the preparation time needed before a method can report its first result. $\mathtt{CSR}_u$ has zero latency, and methods that only restructure the data have lower latency than methods that generate code. Of course, latency varies widely *within* the latter two categories as well.

We tested all methods and libraries on 23 matrices and 5 machines. Our experimental results show two main points:

1. Speedups can be obtained by runtime specialization. In most cases, a method involving runtime code generation is the fastest.
2. There is no one best method: it varies both across machines and across matrices.

Specifically, out of our 115 (23×5) trials, the best specializers were: Stencil (11 times), GenOSKI (29), Unfolding (27), CSRbyNZ (20), CSR (13), MKL (2), BiCSB (6), CSX (7)[2].

The main contribution of this paper is a systematic comparison of a number of methods for performing sparse matrix–dense vector multiplication, including methods that are specialized to a particular matrix. The methods evaluated are "generic" in the sense that they are not designed for matrices of any very particular form, but would apply in general to sparse matrices of the kind found in the Matrix Market [6] and Florida Sparse Matrix Collection [4, 17]. We discuss some of the reasons for the timings we are seeing, including matrix characteristics, and the effect of code and data size and cache size. In addition, we explain how this work fits into the overall goal of creating a matrix-vector multiplication library.

The structure of the paper is this: Section 2 describes in detail the methods we are studying for performing matrix-vector multiplication. Section 3 discusses some aspects of the methods that affect performance. Section 4 describes our experimental setup. Section 5 shows our performance numbers. In Section 6, we discuss how this work might find applications in practice. Section 7 discusses related work; conclusions are presented in Section 8.

2. Methods

In this section, we describe the methods we use. In this discussion, we assume M is an $n \times n$ matrix, with nz non-zeros. We use zero-based indexing for all arrays. The code shown in this section is drawn from the actual generated code.

[1] Potentially, the code for any possible pattern of GenOSKI can be generated offline; however, because there are too many possibilities (e.g. 2^{16} when using 4×4 blocks), opting for runtime generation is likely to be more feasible for this method.

[2] We ran CSX only on 2 platforms due to library conflicts; BiCSB and MKL did not run on one machine because of CPU incompatibility.

2.1 Compressed Sparse Rows

The most common representation for sparse matrices is *Compressed Sparse Rows* (CSR). It consists of three arrays:

- mvalues is an array of floating-point numbers of length nz containing the non-zero values of M in row-major order.
- cols is an integer array of length nz. Element i of this array contains the column number of the i^{th} element in mvalues.
- rows is an integer array of length $n+1$. Element j of this array gives the mvalues-index of the first non-zero element of row j.

With this representation, a standard CSR loop looks as follows (recall that v is the input vector, w is the output vector):

```
for (i = 0; i < n; i++){
  ww = 0.0;
  k = rows[i];  // mvalues[k] = M[i,cols[k]],
                // the first non-zero in row i
  for (; k < rows[i+1]; k++)
    ww += mvalues[k] * v[cols[k]];
  w[i] += ww;
}
```

2.2 CSR Unrolling

$\mathtt{CSR}_u$ partially unrolls the inner loop of the standard CSR method u times, plus inserts a "clean-up" loop handling the leftover elements. The data layout is identical to CSR. Unrolling can produce more efficient code than CSR due to additional instruction level parallelism and reduced loop overhead. However, the difference in performance between CSR and $\mathtt{CSR}_u$ is expected to be small.

2.3 CSRbyNZ

This method groups the rows of M according to the number of non-zeros they contain, and generates one loop for each group. The array rows contains a permutation of the row numbers, in which all the rows with a particular non-zero count are grouped together; cols and mvalues serve the same purpose as with CSR. So, for example, if there are exactly six rows of M that have three non-zeros, the loop for those rows would be:

```
for (i = 0; i < 6; i++, b += 3) {
  row = rows[a++];
  w[row] += mvalues[b]   * v[cols[b]]
         +  mvalues[b+1] * v[cols[b+1]]
         +  mvalues[b+2] * v[cols[b+2]];
}
```

Here, a indexes over rows and b indexes over mvalues. mvalues contains the non-zeros of M in the order in which they are consumed by these loops.

This method gains its efficiency from long basic blocks in each loop, which can be compiled efficiently. It provides, in effect, a perfect unrolling of the inner loop of CSR. CSRbyNZ is similar to the method described by Mellor-Crummey and Garvin [27].

2.4 Unfolding

Unfolding completely unfolds the CSR loop and produces a straight-line program. Despite its simplicity, it needs a detailed explanation as the code it generates has interesting and important implications on the binary code produced by the compiler.

First, recall that this method generates a statement per each matrix row i in the following way:

$\mathtt{w}[i]$ += M_{i,j_0} * $\mathtt{v}[j_0]$ + M_{i,j_1} * $\mathtt{v}[j_1]$ + $\ldots$;

In principle as well as in practice, this method produces the lowest number of dynamic instructions. However, it also produces, by far, the longest code. Yet, surprisingly, in our tests, we have seen that Unfolding occasionally beats the other methods substantially, even for very large matrices. The reason for this is that many matrices have *repeated values*; indeed, the number of distinct values in our sample matrices is usually much less than nz (see Table 1).

This produces speedups for two reasons: reduced memory load, and reduced instructions because of common subexpressions. To see this, suppose there are only three distinct values in the matrix (say, 3, 5, and 9) and let the first two lines of the generated code be

```
w[0] += 9*v[2] + 9*v[3] + 5*v[8] + 3*v[9];
w[1] += 5*v[8] + 3*v[9] + 9*v[11];
```

Having a non-zero value repeated on the *same row* of the matrix allows applying anti-distribution of multiplication over addition (i.e. $c\times v_i + c\times v_j = c\times(v_i+v_j)$). Having the same value repeated on the *same column* of the matrix enables common subexpression elimination (CSE). After applying both optimizations, the above code would look like this:

```
double temp = 5*v[8] + 3*v[9];
w[0] += 9*(v[2] + v[3]) + temp;
w[1] += temp + 9*v[11];
```

The floating point constants are emitted by the compiler — we examined `icc`, `gcc`, and `clang` — into the data section of the object code, and loaded into registers. When the distinct values are very few, registers can be reused to reduce memory loads. In effect, the code above can be compiled as if it were:

```
double M[3] = {9, 5, 3};
double temp = M[1]*v[8] + M[2]*v[9];
double m9 = M[0];
w[0] += m9*(v[2] + v[3]) + temp;
w[1] += temp + m9*v[11];        // m9 reused
```

Unlike all our other methods, and contrary to what we said in the introduction, specialization by this method actually allows a *reduction in the number of floating point operations*.

It is worth mentioning that, although the number of distinct values is usually much less than nz, this fact alone is not that helpful; the number has to be small enough that we are likely to see many repeated values in each row and column, thus allowing the optimizations described. Furthermore, by causing references to matrix values to be accessed out of order — in all other methods, these values are stored in an array that is accessed in strictly sequential order — these optimizations can have a negative effect on locality.

2.5 Stencil

Where `CSRbyNZ` divides up the rows of M according to the number of non-zeros, `Stencil` divides them up according to the exact pattern of non-zeros. Specifically, the "stencil" of each row is defined as the location of non-zeros relative to the main diagonal. So, if row r has non-zeros in columns $r-1$, r, $r+1$, and $r+3$, its stencil would be $\{-1,0,1,3\}$. All the rows that have the same stencil can be handled in a single loop. For example, if rows 2, 4, and 6 are the only ones with stencil $\{-1,0,1,3\}$, then the loop for this stencil is shown below, where the values of M are laid out in the order in which they are consumed by these loops:

```
int stencil_rows[3] = {2, 4, 6};
for (i = 0; i < 3; i++) {
  row = stencil_rows[i];
  vv = v + row;
  w[row] += mvalues[0] * vv[-1] + mvalues[1] * vv[0]
         +  mvalues[2] * vv[1]  + mvalues[3] * vv[3];
  mvalues += 4;
}
```

Like `CSRbyNZ`, `Stencil` gets its efficiency from the long basic blocks inside each loop. `Stencil` also gains an advantage in memory accesses, because it entirely eliminates the `cols` array and the indirect access to `v`. Thus, for matrices with a modest number of stencils, this method can be the most efficient. However, if there are many stencils, the code can get quite large, reducing its efficiency.

2.6 GenOSKI

This method is based on OSKI [19, 21, 31] and is similar to PBR [9]. The idea of OSKI is to divide the matrix into dense blocks (of size, say, $b\times b$) and perform the multiplication on a block basis. By having a loop whose body handles blocks of size $b\times b$, the goal of this optimization is to increase register reuse. It may also reduce the amount of memory required to store indices for the matrix M, since a single pair of indices is stored per block. (For example, if all blocks were perfectly dense, arrays `rows` and `cols` would each be of length nz/b^2, for a total size of $2nz/b^2$, as compared to the total size of $nz+n$ for these arrays in `CSR`.) The drawback of OSKI is that non-empty blocks may still contain zeros, and those have to be added to M explicitly. This increases both the number of floating-point operations and memory communication. The zero fill factor substantially determines whether this method will be efficient. Our experience shows that 1×2, 2×1, and 2×2 blocks are occasionally efficient, but larger blocks almost never are. `GenOSKI` is our attempt to overcome the zero fill problem by generating code.

`GenOSKI` has one loop for each *block pattern* of non-zeros in this matrix. For each pattern, two arrays hold the list of "block locations," the indices of the northwest corner of the blocks that have that pattern. For example, consider a matrix divided into 3×3 blocks and having 18 blocks conforming to the pattern of non-zeros 1,1,0; 1,1,1; 0,1,1: the first two columns on row 0; all three columns on row 2; the second and third columns on row 3. The loop to handle these 18 blocks is shown below. Here `a` and `b` are global variables indexing over blocks and over values, respectively.

```
for (i = 0; i < 18; i++, a++) {
  ww = w + rows[a];
  vv = v + cols[a];
  ww[0] += vv[0]*mvalues[b] + vv[1]*mvalues[b+1];
  b += 2;
  ww[1] += vv[0]*mvalues[b] + vv[1]*mvalues[b+1]
        +  vv[2]*mvalues[b+2];
  b += 3;
  ww[2] += vv[1]*mvalues[b] + vv[2]*mvalues[b+1];
  b += 2;
}
```

`GenOSKI` has low overhead, and indeed often performs well, especially when most blocks are fairly dense. This is a bit surprising, because there are many reasons it should not do so. Zero fill is not a problem *per se*, but it does have an impact: we need to maintain two indexes per block (stored in the arrays `rows` and `cols`), so if there are many sparse blocks, this entails more data than `CSR`. Furthermore, `GenOSKI` can potentially generate a lot of code: for 4×4 blocks, there are 65,535 distinct patterns, which means every 4×4 blocks could have a different pattern. In practice, the number of patterns in a matrix is much smaller than the maximum (Table 1). Lastly, unlike all the other methods, `GenOSKI` does not calculate entire rows at a time, which means that, where the other methods do a single write to each element of `w` — so exactly n writes — `GenOSKI` may do as many as n/b reads *and* writes for each row, or a total of nz/b memory operations on `w`. Nonetheless, as we have noted and will see in Section 5, it often does quite well.

2.7 Latency

In this paper, we are not considering issues of latency, so our remarks here will be very brief. Note that latency comes from the need to re-order data and to generate code. `CSR` and $\mathtt{CSR}_u$ do neither, and have no latency; all other methods do code generation.

`CSRbyNZ`, `Stencil` and `GenOSKI` all involve some kind of analysis prior to code generation: grouping the rows by non-zero count, calculating the stencil of each row, classifying blocks by pattern. In general, we have found that low-level code generation is the most expensive part of the specialization process, and therefore code size

Table 1. Characteristics of the matrices used in the experiments. Matrices labeled (p) are pattern matrices.

Matrix	*n*	*nnz*(M)	*size*(MB)	*nnz/n*	*group*	*#stencils*	*#genOSKI4*	*#genOSKI5*	*#distVals*	*Row_nz*
email-EuAll (p)	265214	0.420	5.82	1.6	SNAP	161683	499	1088	420045	311
cit-HepPh (p)	34546	0.421	4.96	12.2	SNAP	31814	315	683	421578	162
soc-Epinions1 (p)	75888	0.508	6.11	6.7	SNAP	49442	3281	8439	307854	326
soc-sign	77357	0.516	6.21	6.7	SNAP	40649	1212	2867	2	279
web-NotreDame (p)	325729	1.497	18.38	4.6	SNAP	126894	4135	9474	1497134	312
webbase-1M	1000005	3.105	39.35	3.1	Williams	504865	4394	11141	222	370
e40r5000	17281	0.553	6.40	32.0	SPARSKIT	601	130	265	368750	25
fidapm11	22294	0.617	7.16	27.7	SPARSKIT	4682	1197	2576	88275	22
fidapm37	9152	0.766	8.80	83.7	SPARSKIT	8391	876	2102	350166	70
m133-b3	200200	0.801	9.93	4.0	JGD_Homology	200200	489	1627	2	1
torso2	115967	1.033	12.27	8.9	Norris	3148	81	108	806653	3
fidap011	16614	1.091	12.55	65.7	SPARSKIT	7432	1684	3315	211502	71
cfd2	123440	1.604	18.83	13.0	Rothberg	46535	3422	7823	1480984	27
m14b (p)	214765	1.679	20.03	7.8	Dimacs10	172130	3331	9099	1679018	22
s3dkt3m2	90449	1.888	21.96	20.9	CYLSHELL	935	97	143	29116	23
conf6_0-8x8-20	49152	1.917	22.13	39.0	QCD	648	22	156	84553	1
ship_003	121728	1.949	22.77	16.0	DNVS	105098	3982	15702	49424	60
cage12	130228	2.032	23.76	15.6	vanHeukelum	130228	1100	4495	350	28
debr (p)	1048576	2.097	28.00	2.0	AG-Monien	786432	7	9	2097149	3
mc2depi	525825	2.100	26.04	4.0	Williams	2298	50	57	3584	3
s3dkq4m2	90449	2.259	26.20	25.0	CYLSHELL	1131	380	680	8632	29
engine	143571	2.424	28.30	16.9	Dimacs10	84195	108	538	1	147
thermomech_dK	204316	2.846	33.35	13.9	Botonakis	204290	17	329	1967432	9

is the most reliable guide to specialization cost. Size was discussed when presenting the methods: in practice, `Unfolding` produces the longest code, `CSRbyNZ` almost always produces code of modest size (though much bigger than `CSR`), while the amount of code produced by `Stencil` and `GenOSKI` varies by matrix. (We note that when those two methods do produce large codes, they usually do not perform very well.) Performance issues, and their relation to code size, are discussed further in Sections 3 and 5. We do not report any measurements for latency, because the code generator is a python script that we are using to prototype the performance of different methods, not their latency.

2.8 Other Methods

We would like to mention two other potentially useful methods which we are not testing in this study, *vector instructions* and *mixed methods*. In general, our methods cannot efficiently use vector units, due to non-consecutive accesses of vector `v`. For matrices that are almost perfectly banded, elements can be stored in diagonal form, and vector units can be used to advantage. However, in our experiments with this method, it was never the best for our set of matrices. Similarly, regular (non-generative) OSKI never showed well for us. Thus, we do not show results for these two methods.

Another option is to use mixed methods, where a matrix is decomposed into two or more matrices, and each matrix is handled with a different method. For example, we might use the `Stencil` method for the dense bands around the diagonal and `CSRbyNZ` for the remaining elements. We have experimented with this idea, but we have only rarely seen it perform well. Furthermore, the algorithmic space here is so large that it is not yet clear to us how to go about exploring it. For both these reasons, we do not show results for mixed methods here.

3. Performance Issues

In this section we discuss some aspects of the methods that are likely to affect performance; we will return to these in Section 5, after seeing the actual timings.

3.1 Memory Requirements

A significant difference between specialized methods and "generic" methods is that specialization can produce large codes, which can in turn have a major impact on performance. On the other hand, by folding data into the code, the non-code data storage requirements can be reduced. Table 2 contains the expressions to compute code and data size for the various methods. Here we provide some explanation of that table.

`CSR`$_u$: Code size of `CSR`$_u$ is constant, and, for the values of u we consider, small. Data consists of array `mvalues` (nz doubles), array `rows` (n integers), and the `cols` array (nz integers). (Due to a technicality of the representation, `rows` is of length $n + 1$.)

`CSRbyNZ`: Since a different loop is generated for each group of rows with the same count of non-zeros, the code size for `CSRbyNZ` is a function of the number of distinct non-zero counts (Row_nz), as well as the number of non-zeros in each group (nz_row_i). In practice, Row_nz is usually small (Table 1), so code size is modest. Data size is practically the same as `CSR`.

`Unfolding`: For most matrices, `Unfolding` produces the longest code of any of our methods. (In rare cases, `Stencil` can produce code as long; no other method comes close.) As discussed above, repeated values can allow for optimizations that, in some cases, can significantly reduce code size, but this is rare, and in any case still leaves the code very long. (At the very least, the size of the code is O(n), since there is one assignment for each row.) Repeated elements reduce data size significantly in many cases.

`Stencil`: The code size of this method depends on the number of stencils and the size of each stencil. As shown in Table 1, the number of stencils varies widely from matrix to matrix.

`GenOSKI`: The code size for `GenOSKI` is primarily a function of the number of distinct patterns that appear in the matrix. As with stencils, this number varies widely from matrix to matrix (Table 1). In practice, it is always smaller, and usually *much* smaller, than the number of stencils.

3.2 Memory Reference Locality

Another issue affecting performance that will vary by method is locality of memory references. All of our methods except `Unfolding` maintain the values of M in an array of length nz and access it sequentially; there is nothing to be done here about locality. Similarly, the location data in `rows` and `cols` are accessed sequentially. The issue of locality shows up in how the methods reference the input and output vectors `v` and `w`.

Table 2. Expressions to compute code and data size for the different methods.

	CSR	CSRbyNZ	Unfolding	Stencil	GenOSKI
Code Size	c	$\sum_{i=1}^{Row\_nz} nz\_row_i * c$	(possibly) $nz * c$	$\sum_{i=1}^{stencils} nz\_stencil_i * c$	$\sum_{i=1}^{patterns} nz\_pattern_i * c$
Data Size	$nz * 8 + nz * 4 + (n+1) * 4$	$nz * 8 + nz * 4 + n * 4$	$distinct\_nz * 8$	$nz * 8 + n * 4$	$nz * 8 + nblocks * (4 + 4)$

Table 3. Specification of experimental machines.

Name	Processor & Freq (GHz) / Microarchitecture	Cores	Cache Sizes (Bytes)			Mem	OS	icc
			L1 (I/D)	L2	L3	(GB)		
loome2	Intel Core i7 880 @ 3.07 / Nehalem	4	128K	1M	8M	8	Linux CentOS 5.8	14.0
loome3	Intel Core i5 2400 @ 3.10 / SandyBridge	4	32K	256K	6M	8	Linux CentOS 5.8	14.0
i2pc3	Intel Xeon E7-4860 @ 2.27 / Westmere	40	64K	256K	24M	128	Scientific Linux 6.3	14.0
turing	Intel Xeon E5-2620 @ 2.00 / SandyBridge	6	32K	256K	15M	16	Ubuntu Linux 12.04	14.0
milner	AMD FX 8320 @ 3.50 / Piledriver	8	64K/16K	2M	8M	8	ArchLinux	gcc 4.8.2

`CSR`**:** `CSR` maintains perfect locality relative to `w`, as it assigns to its elements sequentially. If M is strongly banded — meaning the non-zeros are clustered around the main diagonal — then it will have good locality in `v` as well. In most cases, there is a dense cluster of non-zeros around the main diagonal, but also a good number of non-zeros elsewhere; in this case, access to `v` will begin to look random, and locality will be poor.

`CSRbyNZ`**:** Here, because of the reordering of rows, access to `w` is no longer sequential. Furthermore, any "natural" locality in `v` — as when a matrix is strongly banded — may be lost. As a consequence, this method does not have particularly good memory behavior relative to either `v` or `w`.

`Stencil`**:** Memory access behaviour of `Stencil` is similar to `CSRbyNZ`. Because each stencil loop may cover rows that are randomly distributed throughout M, and also each stencil contains elements of M potentially randomly distributed throughout a single row, accesses to `v` and `w` are arbitrary.

`GenOSKI`**:** As with all other methods the access to the values, `rows`, and `cols` are perfectly sequential. However, as with `CSRbyNZ` and `Stencil`, accesses to `v` and `w` bear no obvious relation to the natural order, and are likely to be highly non-localized. (Aside from locality issues, we noted earlier that `GenOSKI` performs many more memory operations relative to `w` than the other methods.)

3.3 Parallelization

In this paper, we run all of our codes in parallel, using four threads. It is also interesting to see how these methods perform sequentially, but most researchers are using parallel codes, so parallel times are easier to compare to other methods. For example, we have found that MKL does not perform very well in sequential mode, so that without running it in parallel, comparisons are fundamentally unfair. As another example, `CSX` does not claim to have good performance in the sequential case, but only when parallel execution creates memory contention.

Parallelization of these codes is generally quite straightforward. It is just a matter of splitting M into four horizontal tranches, with approximately equal numbers of elements, applying a method to each, and producing four functions to be run on the four cores. For `CSR` and `Unfolding`, there is really nothing more to it.

For `CSRByNZ`, `Stencil`, or `GenOSKI`, there is one choice to be made before doing the split, and that is whether to sort the rows before splitting. Consider `Stencil`: Suppose M has s stencils, and they are spread throughout the matrix. If we split M into four tranches in the obvious way (what we call "split-by-count"), we are likely to have all s stencils, more or less, show up in each tranche; if there are a lot of stencils, the code running on each processor will be large. If instead we first sort the rows of M by stencil and *then* do the split into four pieces (we call this "split-by-pattern"), each piece will have only a portion of the stencils and will therefore have less code, which is generally better for performance. Note that, for `stencil` and `CSRbyNZ`, we already have to sort the rows into groups, so split-by-pattern is no extra work.

`GenOSKI` presents a somewhat different problem. The method divides the matrix up by patterns, and handles every occurrence of a given pattern in a single loop. If we generate this code first, then assign a subset of the loops to each core, it gives us an even split *and* minimizes code size. However, there is a problem alluded to earlier: any of the patterns can contribute values to any of the rows. If we had code running on separate cores reading and writing to the same location in `w`, we would have to put locks on each one. On the other hand, if we split M into tranches (split-by-count), and generate (sequential) `GenOSKI` code separately in each tranche, there is no need for locks. Although split-by-count results in larger code, the effect is more than offset by avoiding the need for locking.

Accordingly, we parallelize `GenOSKI`, `Unfolding` and `CSR` using split-by-count; `Stencil` and `CSRbyNZ` using split-by-pattern.

4. Experimental Setup

The five target platforms on which we ran our experiments are listed in Table 3. The motivation to select these platforms was to have a variety of LLC cache sizes and microarchitectures. To generate parallel code we used the OpenMP "section" construct and created as many sections as threads. The codes were compiled with icc with the `-O3 -openmp` flags; when icc was not available, we used gcc instead, with the same flags.

We have implemented and evaluated the following methods: `CSR`, $\mathtt{CSR}_u$ with u ranging from 1 to 3, `CSRbyNZ`, `Unfolding`, `Stencil`, and `GenOSKI`. We explained in Section 3.3 why splitting the matrix using split-by-pattern was likely to produce better results for `CSRbyNZ` and `Stencil`, while split-by-count would be better for `GenOSKI`; our experiments confirm this intuition, and accordingly we use the better method in each case and do not report the alternative. (With the split-by-pattern approach, when a loop has to handle more than $nthread$*500 non-zeros, we split the loop to allow for a better balanced workload.) For `GenOSKI`, our experiments show that the best results are obtained with blocks of 4×4 or 5×5, so we only show results for these sizes, and use the names `GenOSKI4` and `GenOSKI5`, respectively.

We compare our methods against the Intel MKL library version 14.0, whenever it is available on the target platform. We also compare against two state-of-the-art SpMV libraries, `BiCSB` [12]

Table 4. Code and Data Size in MB. For `Stencil` and `CSRbyNZ`, we use split-by-pattern. For `GenOSKI`, we use the split-by-count approach. In all the cases, we generate the code for 4 threads.

	CSR		Stencil		GenOSKI4		GenOSKI5		Unfolding		CSRbyNZ	
Matrix	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data
email-EuAll	0.00	5.8	11.5	3.4	0.13	6.5	0.29	6.4	12.2	3.3	1.70	5.9
cit-HepPh	0.00	5.0	9.6	3.4	0.07	6.6	0.22	6.6	9.6	3.3	0.58	5.2
soc-Epinions1	0.00	6.1	12.1	4.1	0.92	7.2	1.90	7.0	12.1	4.0	1.90	6.3
soc-sign	0.00	6.2	11.8	4.1	0.30	7.7	0.62	7.6	5.1	0.0	1.40	6.3
web-NotreDame	0.00	18.4	34.7	12.0	1.50	15.1	2.80	14.6	32.3	12.0	2.70	18.5
webbase-1M	0.00	39.4	77.7	25.0	1.98	32.2	4.00	31.0	31.2	15.9	5.30	41.2
e40r5000	0.00	6.4	0.5	4.5	0.09	5.0	0.30	4.9	12.8	4.0	0.09	6.7
fidapm11	0.00	7.2	5.7	5.0	0.57	6.3	1.34	5.9	14.2	18.9	0.04	7.5
fidapm37	0.00	8.8	19.6	6.1	0.39	7.2	1.34	6.9	20.0	5.9	0.50	9.2
m133-b3	0.00	9.9	19.6	6.4	0.18	10.5	0.84	10.0	9.7	0.0	0.00	10.4
torso2	0.00	12.3	0.6	8.7	0.04	10.1	0.04	9.7	7.4	8.3	0.01	12.8
fidap011	0.00	12.6	13.7	8.8	1.27	9.7	2.74	9.4	25.9	6.0	0.31	13.1
cfd2	0.00	18.8	21.6	13.1	1.81	15.5	4.13	14.9	31.9	11.9	0.04	19.7
m14b	0.00	20.0	38.0	13.5	1.38	22.2	2.85	21.8	39.1	13.4	0.03	20.9
s3dkt3m2	0.00	22.0	0.3	15.5	0.09	16.6	0.20	16.1	39.6	5.5	0.03	23.0
conf6	0.00	22.1	2.0	15.5	0.02	17.3	0.15	17.2	42.1	7.4	0.02	23.2
ship_003	0.00	22.8	44.5	15.6	2.21	19.7	9.52	18.3	43.8	11.8	0.14	23.8
cage12	0.00	23.8	46.6	16.2	0.35	22.4	1.73	22.4	26.6	0.7	0.05	24.9
debr	0.00	28.0	58.1	16.7	0.00	20.9	0.00	20.9	58.4	16.7	0.00	29.3
mc2depi	0.00	26.0	0.4	18.9	0.02	21.9	0.02	20.9	10.8	12.8	0.00	27.3
s3dkq4m2	0.00	26.2	0.8	18.4	0.31	19.8	1.10	19.2	47.0	6.9	0.05	27.4
engine	0.00	28.3	51.8	19.6	0.10	22.8	0.51	22.1	11.8	0.0	0.84	29.6
thermomech_dK	0.00	33.4	64.8	22.7	0.02	28.0	0.12	30.3	63.8	22.3	0.01	34.9

and `CSX` [24], that have online code that can be installed and run. `BiCSB` [2] is implemented on top of CSB [11], a parallel sparse matrix data structure designed for SpMV on multicores. `BiCSB` uses bitmasked register blocks to reduce the memory bandwidth requirement when using register blocking. (We ran both CSB and `BiCSB`, but since `BiCSB` is always faster than CSB, we only compare against `BiCSB`.) `CSX` [3] is based on the Compressed Sparse eXtended format that allows for a flexible storage format to support a variety of structures within the sparse matrix, such as horizontal, vertical, diagonal, anti-diagonal, or blocks. This approach requires runtime code generation. We compare against the SpMV running times, without taking into account the time to generate the code.

Table 1 shows the 23 matrices from the Matrix Market [6] and the University of Florida Sparse Matrix collection [4, 17] that we have used. The table is split into two sets by a horizontal line and each set is sorted by number of non-zeros. The matrices above the horizontal line are derived from graphs that model social or communication networks, among others, and follow a power law distribution. The matrices below the line come from other domains, such as Finite Element modeling (SPARSKIT). Several of these matrices were used in previous studies [11, 24, 34]. We did not select them based on any specific pattern, but rather to represent a variety of domains. Some of these matrices are pattern matrices, for which the source does not provide values; we have generated values for these matrices, with all the generated values being different. The set includes symmetric and structurally-symmetric matrices. For these, we do not apply any shape-based optimization; we simply treat them as regular matrices and, consequently, only report the number of elements that we multiply.

The columns of Table 1 provide the following information: *n* and *nnz*; the size of the CSR code — which is to say, the size of the raw data — in MB; the denseness (*nnz/n*); and the group they come from. The last few columns give data that are useful in evaluating the performance of these methods: *#stencils* is the number of different stencils; *#genOSKI4* and *#genOSKI5* are the numbers of distinct patterns that appear in 4×4 and 5×5 blocks, respectively; *#distVals* is the number of distinct values; and *Row_nz* is the number of distinct row non-zero counts.

Table 4 shows code and data sizes for the matrices for the different methods when we generate OpenMP code for 4 threads. These sizes are drawn directly from the compiled code. Code size values differ slightly from those computed using the expressions in Table 2, as those expressions do not take into account the extra loops that appear when a loop is split for parallel execution into 2 or more threads. Also, the icc compiler unrolls some loops. In addition, to speed up compilation time[3], we split the code into several functions, grouped in multiple files. As a consequence, even if a matrix has a single distinct value, this value will appear once in each file. Thus, for `Unfolding`, the data size in practice is larger than the number of distinct values reported in the table.

To collect the timings, we did the following for each matrix/method/machine combination: (1) Performed matrix-vector multiplication 10,000 times (on an unloaded machine); (2) repeated (1) five times; and (3) chose the fastest of those five runs. For generality, the result of the multiplication is added to the output vector, even though this output vector is zeroed before each multiplication. The 10,000-iteration choice makes time measurements reliable by giving running times that are in the order of seconds even for the smallest matrices in our set. We choose the minimum time in step (3) because it represents the fastest run and the one that is contaminated by the fewest external events. We observed the relative difference of the minimum with respect to the median of the five runs to be usually less than 2%. For instance, in loome2, out of 138 cases (23 matrices $\times$ 6 specializers), only in 5 cases the relative difference is more than 2%, with the maximum being 5%.

5. Experimental Results

In this section, we report our experimental results. We compare with `MKL`, and in broader outline with two state-of-the-art libraries, `CSX` and `BiCSB`. We briefly address the issue of scalability by comparing our methods to others when running on eight threads (rather than our usual four). Finally, we discuss how the characteristics of

[3] Compilers are optimized for human-written code, which tends to be small, so they are slow when compiling large codes produced by a code generator.

Table 5. Best method for all matrices/machines and speedup with respect to `MKL`, except for milner where the baseline is `CSR`. All the methods (including `MKL`) run with 4 threads.

Matrix	loome2	Speedup	loome3	speedup	i2pc3	speedup	turing	speedup	milner	speedup
email-EuAll	CSR	1.48	BiCSB	1.71	CSRbyNZ	1.60	CSRbyNZ	2.05	GenOSKI4	2.11
cit-HepPh	CSRbyNZ	1.21	CSRbyNZ	1.18	Unfolding	1.10	CSRbyNZ	1.30	CSRbyNZ	1.53
soc-Epinions1	CSRbyNZ	1.43	BiCSB	1.48	Unfolding	1.96	CSRbyNZ	2.20	CSRbyNZ	2.17
soc-sign	Unfolding	2.92	Unfolding	2.79	Unfolding	2.80	Unfolding	2.77	Unfolding	3.52
web-NotreDame	GenOSKI4	1.12	CSR	1.03	Unfolding	1.28	GenOSKI4	1.23	GenOSKI4	1.57
webbase-1M	Unfolding	1.33	Unfolding	1.39	Unfolding	1.74	Unfolding	1.31	GenOSKI4	1.38
e40r5000	Stencil	1.35	Stencil	1.89	CSR	1.02	CSR	1.05	GenOSKI4	1.73
fidapm11	CSRbyNZ	1.07	CSX	1.20	CSR	1.03	CSR	2.11	CSRbyNZ	1.40
fidapm37	GenOSKI4	1.28	CSX	1.35	CSR	1.08	CSR	1.06	GenOSKI4	1.38
m133-b3	Unfolding	1.19	Unfolding	1.22	Unfolding	1.40	CSRbyNZ	1.91	CSRbyNZ	1.50
torso2	Stencil	1.76	GenOSKI5	1.47	Unfolding	1.58	Unfolding	2.08	GenOSKI5	2.03
fidap011	CSX	1.47	GenOSKI4	1.11	CSR	1.04	CSRbyNZ	1.49	GenOSKI4	1.45
cfd2	CSX	1.19	CSX	1.18	Unfolding	1.08	GenOSKI4	1.09	GenOSKI4	1.55
m14b	CSRbyNZ	1.42	BiCSB	1.64	CSR	1.27	BiCSB	1.56	CSRbyNZ	1.43
s3dkt3m2	Stencil	1.63	Stencil	1.51	Stencil	1.19	Stencil	2.16	GenOSKI5	2.04
conf6_0-8x8-20	Stencil	1.39	GenOSKI4	1.41	CSR	1.01	GenOSKI4	1.62	GenOSKI4	1.76
ship_003	CSR	1.07	CSX	1.20	MKL	1.00	CSR	1.01	GenOSKI4	1.31
cage12	CSX	1.22	CSRbyNZ	1.08	Unfolding	1.56	GenOSKI4	1.10	GenOSKI4	1.28
debr	CSRbyNZ	1.15	CSRbyNZ	1.09	GenOSKI4	1.49	BiCSB	1.15	CSRbyNZ	1.36
mc2depi	Unfolding	1.29	Unfolding	1.23	Unfolding	1.61	GenOSKI5	1.52	GenOSKI5	1.76
s3dkq4m2	Stencil	1.51	Stencil	1.45	MKL	1.00	Stencil	1.81	GenOSKI5	1.73
engine	Unfolding	3.24	Unfolding	2.85	Unfolding	3.89	Unfolding	6.20	Unfolding	1.82
thermomech_dK	GenOSKI4	1.11	BiCSB	1.06	GenOSKI4	1.01	GenOSKI4	1.41	GenOSKI4	1.61
Average		1.47		1.46		1.46		1.78		1.71

Table 6. Comparison between methods.

		loome2	loome3	i2pc3	turing	milner
`CSR`	Avg. Speedup	1.12	1.09	1.08	1.15	-
	# matrices is best	2	1	6	4	-
	# matrices is better	23	20	16	20	-
	Avg. Speedup if better	1.12	1.11	1.12	1.17	-
`Stencil`	Avg. Speedup	0.79	0.81	1.01	0.81	0.93
	# matrices is best	5	3	1	2	0
	# matrices is better	6	6	11	6	7
	Avg. Speedup if better	1.49	1.46	1.32	1.68	1.6
`GenOSKI4`	Avg. Speedup	1.06	1.15	1.05	1.28	1.58
	# matrices is best	3	2	2	5	11
	# matrices is better	14	16	11	19	23
	Avg. Speedup if better	1.21	1.29	1.25	1.37	1.58
`GenOSKI5`	Avg. Speedup	0.97	1.04	0.98	1.19	1.50
	# matrices is best	0	1	0	1	4
	# matrices is better	10	9	10	15	23
	Avg. Speedup if better	1.22	1.31	1.21	1.38	1.50
`Unfolding`	Avg. Speedup	0.88	0.82	1.30	1.03	0.84
	# matrices is best	5	5	11	4	2
	# matrices is better	6	6	13	6	4
	Avg. Speedup if better	1.89	1.78	1.75	2.59	2.05
`CSRbyNZ`	Avg. Speedup	1.13	1.14	1.12	1.29	1.32
	# matrices is best	5	3	1	5	6
	# matrices is better	20	22	12	17	23
	Avg. Speedup if better	1.16	1.15	1.28	1.43	1.32
`CSX`	Avg. Speedup	0.86	0.90	-	-	-
	# matrices is best	3	4	-	-	-
	# matrices is better	8	10	-	-	-
	Avg. Speedup if better	1.29	1.20	-	-	-
`BiCSB`	Avg. Speedup	0.66	1.07	0.63	0.99	-
	# matrices is best	0	4	0	2	-
	# matrices is better	2	10	3	11	-
	Avg. Speedup if better	1.04	1.30	1.26	1.22	-
Best Specialization	Avg. speedup	1.47	1.45	1.44	1.77	1.71
	#matrices is better	23	23	17	20	23
	Avg. Speedup if better	1.47	1.45	1.62	1.84	1.71

the machines and matrices help explain the timing results; the latter is important in the process of predicting the best method.

5.1 Comparison of Methods

Table 5 shows, for each matrix and machine, the best method among `MKL`, `CSR`, `Stencil`, `GenOSKI4`, `GenOSKI5`, `Unfolding`, `CSRbyNZ`, `CSX` and `BiCSB` (we only consider `CSR` because `CSR` and $\mathtt{CSR}_u$ are always very close to each other). The table also shows the speedup with respect to `MKL`, where the speedup is computed by dividing the `MKL` running times by the running times of each method, when all run with four threads (including `MKL`). For milner, we could not run `MKL`, because it has an AMD processor. In addition, the AMD's Core Math Library (CML) does not have level-2 sparse operations [1]. Hence, for this machine we compare with `CSR`. The last row shows, for each machine, the average speedup obtained when the best method is used. We were able to run `CSX` only on loome2 and loome3 due to library conflicts. Also, we could not run `BiCSB` on milner because `BiCSB` requires the icc compiler.

Table 6 compares the different methods. For each method and machine the table shows the average speedup if that method is used for all the matrices, the number of matrices for which that method is the best, the number of matrices that run faster than `MKL` using that method, and the average speedup of that method if only used when it runs faster than `MKL`. The last two metrics tell us how often each method improves with respect to `MKL`, and if it improves, what is the average speedup. The last row in the table (labeled Best) shows the same metrics, but when the best specializer is chosen. In this case, "Avg. speedup" is the speedup obtained if we always use a method that requires specialization (`CSR`, $\mathtt{CSR}_u$, `MKL`, and `BiCSB` do not require specialization, while all the others do); in some cases this will result in slowdowns with respect to `MKL`. This value is very similar to the Avg. speedup of the best method, shown in the last row of Table 5.

Overall, the results show that specialization can produce significant speedups. Out of 23 matrices, specialization produces speedups for 23, 23, 17, 20, and 23 matrices and average speedups of 1.47, 1.45, 1.44, 1.77, and 1.71 for loome2, loome3, i2pc3, turing, and milner, respectively. The average speedups are computed using the best specialization method, even if this method is slower than a method that does not require specialization.

Figures 1 and 2 show MFLOPS/sec for every combination of machine/matrix/method. To reduce clutter, we omit `CSR` and `GenOSKI5`. The figures show that our methods are usually faster than `CSX` and `BiCSB`. Moreover, when one of these methods is faster, the difference with one of our methods is usually very small.

We have also run some experiments to evaluate the scalability of our methods. Figure 3 shows speedups for `MKL` with 4 and 8 threads, and the best of `Unfolding`, `CSRbyNZ`, `Stencil`, and `GenOSKI4` when running with 4 and 8 threads with respect to `MKL` with a single thread. The figure shows that in all cases, but in three matrices, a method that requires specialization is better than `MKL`. It also shows that the methods that require specialization scale well.

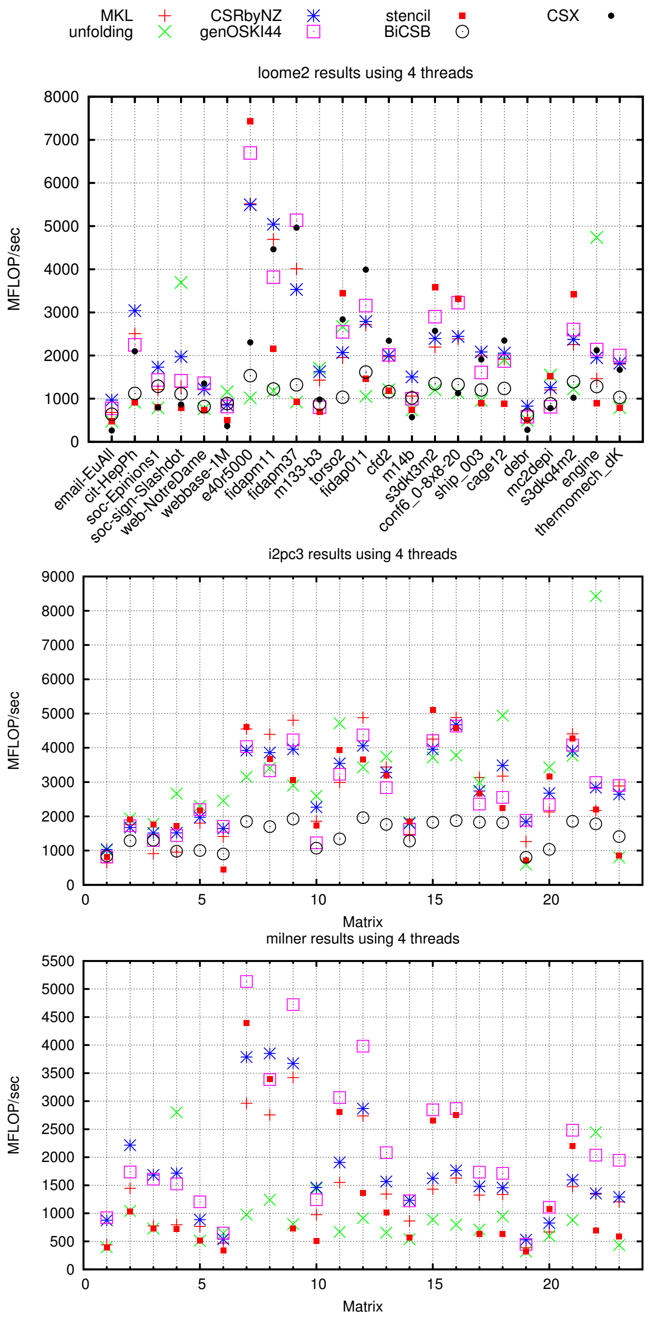

Figure 1. MFLOP/sec for loome2, i2pc3 and milner.

5.2 Explaining the Timings

The natural question is how to determine what is the best method. Our results show that speedups depend on both machine and matrix characteristics. For two matrices (soc-sign and engine), the same method is the best across the board. For the rest, the best method varies across machines. For instance, for email-euAll, there are four different methods with very different speedups. We now discuss how the machine and matrix characteristics (Tables 1, 3, and 4) help explain the timings (Tables 5 and 6).

`CSR` or $\mathtt{CSR}_u$ are not usually the best methods. Although not shown, running times for $\mathtt{CSR}_u$ are very similar to those of `CSR`.

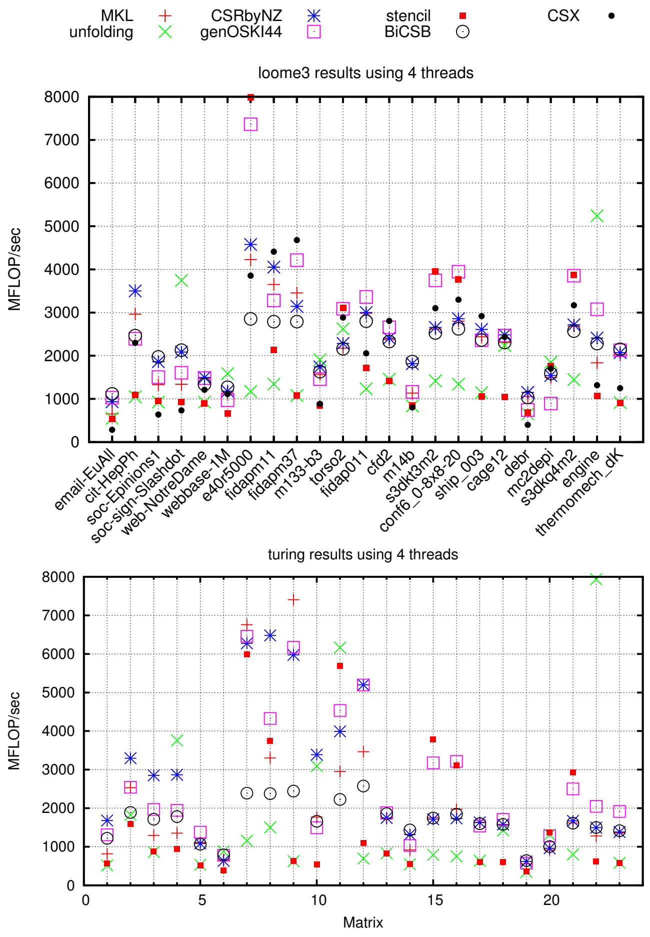

Figure 2. MFLOP/sec for loome3 and turing.

`Unfolding` is the best method when the sum of code and data size fits in the Last Level Cache (LLC) (Table 3 and 4). Our matrices are large, and this term should always be large. However, as explained in Section 2, when the number of distinct values is small (*distVals* in Table 1), the compiler can apply certain optimizations such as CSE, that significantly reduce the code size. Matrices that benefit from this method across the board are soc-sign and engine. soc-sign and engine have only 2 and 1 distinct values, respectively, and achieve significant speedups in all the platforms. For webbase-1M, the number of distinct values is 222, but it is our largest matrix in terms of non-zero elements, and thus `Unfolding` is the best for all machines except milner. For m133-b3, `Unfolding` is the best in all platforms except turing and milner (on these two platforms, `Unfolding` is the second best method with only a very small difference between them). m133-b3 obtains, in general, lower speedups than soc-sign and engine, even though it only has 2 distinct values. The reason is that the code size of `Unfolding` for m133-b3 is about the size of the `CSR` data. The results also show that for i2pc3, `Unfolding` is the best method for 11 matrices. This is because i2pc3 has the largest LLC (24MB). To the best of our knowledge, this is the first study that reports the benefit of `Unfolding` when the number of distinct values is small. This can be applicable to a large set of matrices, like those derived from graphs, such as the adjacency matrix or laplacian matrix. Another example are algebraic multigrid methods for sparse linear systems [10].

`Stencil` has the potential to produce good speedups, but only the matrices with a small number of stencils can benefit from it.

`Stencil` is not the best method across the board for any matrix, but it is the best for s3dkt3m2 for all machines, except milner. s3dkt3m2 has 935 different stencils, which results in larger code compared to the 94 different patterns of `GenOSKI4` (see Table 1). However, `Stencil` is more efficient than `GenOSKI` in all the machines (except milner, where `GenOSKI` is usually the best method) because code size is relatively small, it does not require the `cols` array or indirect access to `v`, and it has good locality in the output vector. `Stencil` is also usually good for s3dkq4m2, torso2, mc2depi, and e40r500. Although `Stencil` might not be the best method for these matrices, is usually as good as the best. Notice that these matrices (together with conf6_0-8x8-20) are the matrices with the smallest number of stencils. For conf6_0-8x8-20, that has only 648 stencils, `GenOSKI` is better in loome3, turing, and milner, but `Stencil` is the second best method on those machines. `Stencil` delivers significant speedups, when it is better than `MKL`, as shown in Table 6.

`CSRbyNZ` always produces small codes. Even for the power law matrices (matrices from the SNAP group and webbase-1M) that have a relatively large *Row_nz* (see Table 1), it is still much smaller than the number of stencils or block patterns. The data size of this method is similar to that of `CSR`, but the code executes fewer loop overhead instructions, resulting in higher Instruction Level Parallelism (ILP). This method tends to have modest speedups, although it produces significant speedups on turing. We consider this to be a default method that can be used when none of the other methods seems appropriate. It is interesting to notice that many of the power law matrices benefit from this method in loome2, turing, and milner machines, which have smaller caches than i2pc3.

`GenOSKI` always produces modest code size (see Table 4), as the number of patterns is never too big: out of 65,536 possible patterns when using blocks of size 4×4, the maximum in Table 1 is 4,394. However, the number of patterns is not the most important feature to determine the performance of this method, as it is the best method for web-NotreDame, which has 4,135 patterns (only webbase-1M has more) and is not the best method for debr, that has the lowest number of patterns, only 7. The ability of this method to decrease data size is also important, and that depends on the number of blocks that are empty (each block needs a `cols` and a `rows` index) and the locality. Overall, `GenOSKI` produces good results most of the times for 6 matrices: web-NotreDame, fidapm37, fidap011, cfd2, conf6_0-8x8-20, and thermomech_dK. Speedups of this method are comparable to those of `CSRbyNZ`. milner stands out as the machine most favorable, by far, to `GenOSKI`. (4×4 is usually the best block size; 5×5 is occasionally better. We have also evaluated smaller blocks, but we do not report results, as they are never better.)

`CSX`, which does specialization, is the best method for 3 matrices on loome2 and 4 matrices on loome3. It seems to perform well on matrices where `GenOSKI` is also a good option. `BiCSB` is the best only for 4 matrices on loome3, and 2 matrices on turing. Similarly, `MKL` is the best for 2 matrices on i2pc3.

6. Applications

Knowing that efficient codes can be produced by code generation is interesting, but is it useful? That depends entirely upon the tolerance for latency in the particular application.

We note that it is very common for the *shape* of a matrix — the exact locations of its non-zeros — to be known even when the values are not. Some of these are referred to as "pattern matrices," and the Matrix Market and the Florida collection include many of them. Also, for those matrices derived using Finite Element methods [26], the shape of the matrix may be known ahead of time, as the matrix is derived from a mesh that is usually available before solving the problem. All of our methods except `Unfolding` generate code based only on the shape; by generating code for those matrices off-line — only the `mvalues` array needs to be supplied at runtime — the issue of latency is entirely obviated.

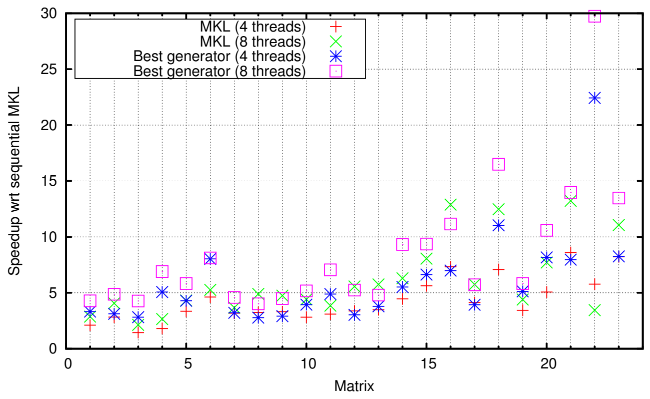

Figure 3. Speedup versus sequential `MKL` for 4 and 8 threads on i2pc3.

The more challenging case is when nothing is known about the matrix until runtime. The work presented here is the first step in the creation of a library for matrix-vector multiplication that will use runtime specialization, *auto-tuning*, and *machine learning techniques* to predict the best method, as has been done in previous work [19, 20, 25, 28, 33]. The library would be employed in cases where a single matrix M is to be multiplied by many vectors.

Here is how we envision the library working: The user will supply the matrix to the library, and the library will produce a pointer to a function of type `void multByM (double v[], double w[])`. When called subsequently, `multByM` will multiply M by `v` and place the result in `w`. (The OSKI library [19, 21, 31] operates similarly.) When first presented with M, the system will determine which method will produce the most efficient `multByM`. It may determine that `CSR` is the best, and will immediately return a pointer to pre-existing code; or it may determine that a specialized code, which must be generated at runtime, will be most efficient. This process itself will take time, and generating the specialized code, if that is the decision, will take even more; in any case, the system cannot produce overall speedups if the matrix is to be multiplied only a small number of times. (The risk might be managed by running program generation in parallel with a low-latency method like `CSR` until the generated code is ready.)

This library organization raises several questions:

1. What methods of generating `multByM` are likely to produce efficient code and what are the speedups that these methods can deliver? This is the question we address here.
2. How can the system determine the best method for a particular matrix on a particular machine?
3. How can the latency introduced by the code specialization process be minimized?

Question (2) will be addressed by auto-tuning [19, 20, 25, 28, 33]. Here, one gathers information about the machine at "install time," and feeds it into the runtime specialization process, which uses it, together with characteristics of the matrix M, to determine how best to generate `multByM`. To minimize latency (question 3), we are developing purpose-built code generators.

7. Related Work

Sparse matrix-dense vector multiplication is an operation that is used in many scientific problems. It has been studied in the OSKI project [21]. A number of researchers have looked at multi-core implementations [9, 11, 12, 18, 22, 24, 27, 34]. Among those, we have compared our codes with CSB, BiCSB [12], and CSX [24], as their libraries were available online. CSRByNZ is similar to the method described by Mellor-Crummey and Garvin [27], while GenOSKI is similar to PBR [9]. Perhaps, the main difference between our work and previous ones, is that rather than evaluating a single method, we are evaluating many. Our goal was to understand if, and by how much, specialization could improve performance.

As discussed in Section 6, auto-tuning is used to overcome the problem that the best code for a problem can vary from machine to machine. It is used by OSKI; other examples are [20, 25, 28, 33].

The area of program specialization — also called *code generation*, *partial evaluation*, or *staging* — has been quite heavily studied, especially with respect to language features, such as type-checking, that promote simplicity and safety of specialization [13, 30, 32]. Work in this area specifically addressing high-performance for realistic applications includes work on marshalling [7, 14] and on code-optimizing transformations [15]. With *runtime* specialization, the focus moves toward the efficiency of specialization itself [23, 29].

8. Conclusions

In this paper we have shown that specialization can be used to obtain speedups for SpMV. Our experimental results using 23 matrices and five machines show that a method requiring specialization runs faster than MKL in 106 out of 115 runs (23×5). These experimental results include comparisons with state of the art libraries, such as Intel's MKL, BiCSB, and CSX. If we only use specialization, the average speedup with respect to Intel's MKL library ranges from 1.44x to 1.77x, depending on the machine. For individual matrices, these speedups can be higher.

In this paper, rather than evaluating a single method, we are evaluating many. Our results show that there is no one best method and that the best method depends on the machine and matrix characteristics. Among the evaluated methods, we have found that one of our methods, Unfolding, can produce significant speedups when the number of distinct values is small. This is important, as this can be common in matrices that are derived from graphs, such as the Laplacian matrix, or algebraic multigrid methods for sparse linear systems.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Award CCF 1017077, and The Scientific and Technological Research Council of Turkey under Grant 110E028. We also would like to thank Edgar Black and Asım Yıldız for their help in the earlier stages of this work, to Luke Ohlson for the fruitful discussions, and to the reviewers for their comments.

References

[1] AMD Core Math Library. http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml.

[2] CSB library. http://gauss.cs.ucsb.edu/∼aydin/csb/html/index.html.

[3] CSX library. https://github.com/cslab-ntua/csx.

[4] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[5] MKL. http://software.intel.com/en-us/articles/intel-mkl/.

[6] Matrix Market. http://math.nist.gov/MatrixMarket/.

[7] B. Aktemur, J. Jones, S. Kamin, and L. Clausen. Optimizing marshalling by run-time program generation. In *GPCE '05*, pages 221–236, 2005.

[8] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shonan challenge for generative programming. In *PEPM '13*, pages 147–154, 2013.

[9] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *ICS'09*, pages 100–109, 2009.

[10] N. Bell, S. Dalton, and L. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[11] A. Buluç, J. Fineman, M. Frigo, J. Gilbert, and C. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA'09*, pages 233–244, 2009.

[12] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *IPDPS '11*, pages 721–733, 2011.

[13] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *POPL '11*, pages 81–92, 2011.

[14] A. Cohen and C. Herrmann. Towards a high-productivity and high-performance marshaling library for compound data. In *2nd MetaOCaml Workshop*, 2005.

[15] A. Cohen, S. Donadio, M. J. Garzarán, C. Herrmann, O. Kiselyov, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, 2006.

[16] R. Davies and F. Pfenning. A modal analysis of staged computation. In *POPL '96*, pages 258–270, 1996.

[17] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[18] E. D'Azevedo, M. Fahey, and R. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *ICCS'05*, pages 99–106, 2005.

[19] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self Adapting Linear Algebra Algorithms and Software. *Proc. of the IEEE*, 93(2):293–312, 2005.

[20] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI '99*, pages 169–180, 1999.

[21] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1): 135–158, 2004.

[22] A. Jain. pOSKI: An extensible autotuning framework to perform optimized SpMVs on multicore architectures. Master's thesis, U. of California at Berkeley, 2008.

[23] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: Run-time Code Generation for Java and Its Applications. In *CGO '03*, pages 48–56, 2003.

[24] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. Csx: An extended compression format for spmv on shared memory systems. In *PPoPP'11*, pages 247–256, 2011.

[25] X. Li, M. J. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms . In *CGO '05*, pages 99–110, 2005.

[26] A. Logg, K.-A. Mardal, and G. N. Wells. Automated solution of differential equations by the finite element method (chapter 6). https://bitbucket.org/fenics-project/fenics-book/downloads.

[27] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. *Int. J. High Perform. Comput. Appl.*, 18(2), 2004.

[28] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232–275, 2005.

[29] F. Smith, D. Grossman, G. Morrisett, L. Hornof, and T. Jim. Compiling for template-based run-time code generation. *J. of Functional Programming*, 13(3):677–708, 2003.

[30] W. Taha and M. Nielsen. Environment classifiers. In *POPL '03*, pages 26–37, 2003.

[31] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Supercomputing '02*, page 26, 2002.

[32] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *PLDI '10*, pages 400–411, 2010.

[33] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Sofware and the ATLAS Project. *Parallel Computing*, 27 (1-2):3–35, 2001.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing'07*, pages 38:1–38:12, 2007.