Optimization of Sparse Matrix-Vector Multiplication by Specialization

Abstract

Program specialization is the process of generating optimized programs based on available inputs. It is particularly applicable when some input data are used repeatedly while other input data vary. Specialization can be employed at compile-time as well as at run-time, depending on when the inputs become available.

In this paper we explore the potential for obtaining speed-ups for sparse matrix-dense vector multiplication using specialization, in the case where a single matrix is to be multiplied by many vectors. We experiment with three methods involving matrix-based specialization, comparing them to four methods that do not (including OSKI and INTEL's MKL library). For this work, our focus is the evaluation of the speed-ups that can be obtained with program specialization without considering the overheads of the code generation. Our experiments use 17 matrices from the Matrix Market and run on seven machines. In 100 of those 119 cases, the specialized code runs faster than any version without specialization. We have also found that the best method depends on the matrix and machine; no method is best for all matrices and machines. Our results also show that hybrid methods may also improve the performance further.

Keywords sparse matrix-vector multiplication, program specialization, run-time code generation.

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)?

Program specialization is a well-studied area. It also goes by the name *staging* [Taha and Sheard, 2000; Westbrook et al., 2010]; a popular technique of *automatic* specialization is called *partial evaluation* [Jones et al., 1993]. 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. It is easy to see that 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 [Davies and Pfenning, 1996]). This code is "optimal" in the sense of producing the minimum instruction count; a standard Compressed-Sparse-Row (CSR) loop (see §2.1) 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 §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 CSR_u.

Diagonal. This is a well-known method that is applicable when M is very strongly banded — that is, a large percentage of its non-zeros occur on diagonals which are very dense. It uses a simple loop that can be easily vectorized on modern computers. All our calculations use double-precision floating-point numbers on machines with 128-bit wide vector registers, so we are limited to a 2-fold speed-up at best. (Also, when diagonals are not 100% dense, zeros need to be inserted, increasing the number of floating-point operations and loads.)

OSKI. This is the method implemented in the OSKI library [Im et al., 2004; Demmel et al., 2005; Vuduc et al., 2002] that takes a matrix and divides it into blocks to generate efficient per-block code (at the cost of having to insert some zeros into the matrix data). $\text{OSKI}_{r,c}$ is OSKI with blocks of size $r \times c$.

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, using a representation similar to Mellor-Crummey and Garvin [2004]. 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.

The methods can be classified into four groups:

- Those that are completely generic and operate on the standard CSR representation (CSR).
- Those that require some restructuring of the data but are still generic and do not perform matrix-based specialization of code (OSKI, diagonal).
- Those that require specialization based on the locations (but not on actual values) of the non-zero elements of the matrix (CSRbyNZ, stencil).
- Those that require specialization based on the non-zero values in the matrix (unfolding).

The "bottom line" is shown in Table 1. We tested the methods on seventeen matrices and seven machines. This table shows which method was most efficient for each pair, and says how much it improved over CSR_2 (chosen as the most efficient non-specialized method). This table illustrates the two main points of this paper:

1. In most cases, one of the methods involving matrix-based code generation is the fastest.
2. There is no one best method: it varies both across machines and across matrices.

Specifically, the table shows that the simple unfolding described above is indeed sometimes the fastest method. In fact, the simple CSR code (with some unrolling of the inner loop) is also, sometimes, the fastest method. Out of our 119 (17×7) trials, the best specializers were: stencil (56 times), CSRbyNZ (29), unfolding (15), diagonal (7), CSR_3 (7), CSR_4 (2), CSR_5 (3).

We have also run experiments using hybrid methods, where we, for example, use a stencil calculation for some subset of the matrix and an unrolled CSR loop for the remainder. The space of possibilities here is so huge that we explore it only superficially, although we present what we believe will be the best hybrid methods (see §4.1).

The main contribution of this paper is a systematic comparison of a number of methods for performing sparse matrix–dense vector multiplication, including

	chicago	i2pc4	loome1	loome2	loome3	pl	turing
add32	CSRbyNZ(65)	CSRbyNZ(91)	CSRbyNZ(88)	CSRbyNZ(88)	CSRbyNZ(88)	CSRbyNZ(69)	CSRbyNZ(94)
cavity02	CSRbyNZ(81)	stencil(70)	CSR_5(51)	unfolding(71)	CSR_3(95)	CSRbyNZ(82)	CSR_4(92)
cavity23	stencil(72)	stencil(75)	stencil(76)	stencil(76)	CSR_3(97)	stencil(73)	CSR_3(96)
fidap001	CSRbyNZ(76)	stencil(67)	unfolding(65)	unfolding(67)	CSR_3(97)	CSRbyNZ(80)	CSR_3(92)
fidap005	unfolding(27)	unfolding(41)	unfolding(41)	unfolding(41)	unfolding(51)	unfolding(34)	unfolding(50)
fidap031	stencil(60)	stencil(60)	stencil(75)	stencil(75)	CSR_3(95)	stencil(64)	CSR_3(95)
fidap037	CSRbyNZ(78)	CSRbyNZ(85)	CSRbyNZ(88)	CSRbyNZ(87)	CSR_4(92)	CSRbyNZ(81)	CSR_5(91)
memplus	CSRbyNZ(77)	CSRbyNZ(90)	CSRbyNZ(91)	CSRbyNZ(92)	CSRbyNZ(93)	CSRbyNZ(80)	CSRbyNZ(96)
mhd3200a	stencil(60)	stencil(66)	stencil(66)	stencil(63)	stencil(86)	stencil(62)	stencil(85)
mhd4800a	stencil(60)	stencil(65)	stencil(57)	stencil(63)	stencil(85)	stencil(61)	stencil(87)
nnc666	CSRbyNZ(71)	unfolding(50)	unfolding(48)	unfolding(49)	CSRbyNZ(84)	CSRbyNZ(74)	CSRbyNZ(84)
orsreg1	stencil(63)	stencil(68)	stencil(40)	stencil(69)	stencil(64)	stencil(66)	stencil(68)
pde900	diagonal(52)	diagonal(53)	stencil(37)	diagonal(53)	diagonal(49)	diagonal(57)	diagonal(53)
rdb2048	CSRbyNZ(94)	unfolding(62)	CSR_5(59)	unfolding(69)	stencil(66)	CSRbyNZ(93)	stencil(67)
saylr4	diagonal(66)	stencil(64)	stencil(86)	stencil(64)	stencil(64)	stencil(61)	stencil(69)
sherman5	stencil(50)	stencil(60)	stencil(46)	stencil(63)	stencil(69)	stencil(53)	stencil(72)
utm5940	stencil(59)	stencil(68)	stencil(69)	stencil(70)	stencil(84)	stencil(61)	stencil(81)

Table 1: Best method for all matrices/machines; run time given in parentheses as a percentage of CSR_2. Characteristics of the machines and matrices can be found in Tables 3 and 4.

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 [Matrix Market]. (It is always possible that a better method than any of ours could be found for a *specific* matrix, using more subtle properties of that matrix.)

Our experimental results show that methods that require matrix-based specialization usually have the best performance. These methods only make sense in a scenario like the one described above, where one matrix is multiplied several times. Depending on when the matrix becomes available, specialization may take place off-line or at run-time. In this work, *we are not considering the cost of specialization*; rather, the question that we are first trying to answer is whether specialization can produce speedups, and if so, of what magnitude. Of course, the time devoted to specialization, especially when employed at run-time, is a major issue which we discuss in §5.

Our experiments include comparisons with the running times of the Intel math kernel library [Intel MKL]. We discuss some of the reasons for the timings we are seeing, including matrix characteristics, and the effect of code size and instruction 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: In §2, we explain the set of methods listed above in more detail; §3 describes our experimental setup; §4 shows our performance numbers. In §5 we discuss how the methods can be used in the context of a library. §6 discusses related work. Finally, future work and conclusions are presented in §7.

	Indirect access to v?	Zero filling?	Code Size
CSR	Yes	No	small
Unrolled CSR	Yes	No	small (proportional to degree of unroll)
Diagonal	No	Yes	small
OSKI	Yes, but fewer than CSR	Yes	small (proportional to block size)
Unfolding	No	No	large (proportional to # of nonzeros)
CSRbyNZ	Yes	No	modest (proportional to unique row lengths)
Stencil	No	No	modest to large (proportional to # of stencils)

Table 2: Some characteristics of the methods

2 Methods

In this section, we describe the methods we use — which were briefly described in the introduction — in detail.

Here we are describing the basic methods, and most of our timing results (such as those in Table 1) involve just basic methods. It is possible that the most efficient code for M is obtained by *mixing* methods; this is discussed in §4.1.

Before presenting the methods, we briefly mention the performance issues that arise with any method:

Instruction count. This metric refers to the dynamic number of instructions. All methods execute the same set of floating-point operations (possibly in a different order), except diagonal and OSKI, which may perform extra computations, due to the extra zeros added to obtain dense diagonals or blocks, respectively. However, methods vary in the number of integer and control operations they execute. One source of variation is this: Because different columns in each row are occupied, different elements of the input vector are accessed in each row; the obvious approach is to access elements of v indirectly, through an array of indices; however, some of our methods can avoid this.

Memory reference locality. Some methods require that M be restructured, but once that is done, its values are referenced in purely sequential order, in a single pass, for each call to `multByM`. However, the order in which elements of v and w are accessed, and the data locality achieved in each, varies. This affects the number of cache misses in reading/writing those values.

Code size. The code size can affect performance by adding to the number of memory reads, especially if the code fails to fit into the instruction cache. The standard generic code — the CSR code and its unrollings — is very short, and once loaded is unlikely to produce any cache misses. The unfolded code produces code proportional in length to the number of non-zeros in M. The stencil code falls in the middle; the length of the

code produced by this method depends on the number of stencils in the matrix.

Table 2 characterizes each method in terms of code size, and whether indirect access to v or filling (adding zeroes to M) is required.

As has been observed many times, modern CPUs defy simple modelling — they often seem to vary in performance almost at random. The precise impact of these factors has so far proven difficult to determine. Still, it is good to keep them in mind in assessing how efficient a given method is likely to be.

In this discussion, we assume M is an $n \times n$ matrix, with nz non-zeros. Also, we use zero-based indexing for all arrays.

(The code shown in this section is drawn from the actual generated code. For most methods, we experimented with several versions of the code and adopted the best. For OSKI, the results are based on our own coding of the method, because experiments showed that our code generally outperformed the OSKI library. A detailed discussion of low-level coding issues appears on our web page [Authors' Web]. The Python script that generates the code is available there as well.)

2.1 Compressed sparse rows

The most common representation for sparse matrices is *compressed sparse rows*, or *CSR*.

The representation consists of three arrays:

- `mvalues` contains the non-zero values of M, in row-major order; this is a double array of length nz.
- `columns` contains the indices of the columns in which non-zeros occur in each row, again in row-major order; this is an integer array of length nz.
- `rows` is an integer array of length $n+1$; for each $0 \leq i \leq n-1$, `rows[`i`]` gives the index in `mvalues` and `columns` at which the non-zeros for row i begin. `rows[`n`]` contains nz; thus, for each i, `rows[`$i+1$`] - rows[`i`]` is the number of non-zeros in row i.

With this representation, the inner loop of `multByM` is this, where `i` ranges from 0 to $n-1$:

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

Note that this is generic code, not dependent in any way on M, and therefore not requiring run-time specialization. We have found that unrolling the inner loop, to a point, can be helpful. CSR_u is CSR with an unrolling factor of u;

"CSR" means CSR_1. If $u > 1$, the code consists of an unrolled loop handling u elements at a time and a "clean-up" CSR_1 loop handling the leftover elements.

CSR has a fairly high instruction count. Code size, even when the inner loop is unrolled, is small. It should have good locality relative to w, referencing each element exactly once. As to v, it depends. If M is strongly banded — meaning the non-zeros are exclusively 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.

For both CSR and unfolding, we have experimented with blocking to improve locality in v, but this has failed to produce any efficiency improvements. Most likely, the matrices we are finding in the Matrix Market are banded enough that there is naturally good locality in v.

2.2 Diagonal

If a matrix is almost perfectly banded, then this method can be used to take advantage of vector units on many machines. In our case, since our calculations are done in double-precision, and we are running the experiments in machines with 128-bit wide vector registers the potential speed-up factor is 2 (or slightly less, due to overheads).

The idea is to represent the matrix by diagonals (instead of rows) and multiply each diagonal by part of v and add those values to the corresponding part of w. For example, for the main diagonal, the loop iterates over indices 0 to $n-1$ and at each iteration executes "`w[i] += mvalues[i]*v[i]`" — code that is easily vectorized.

Vector units can operate only on consecutive values, so this method requires that any missing zeros be included in the representation, detracting from its efficiency. Thus, we use this method only for matrix diagonals that are at least 80% dense; elements that do not occur in dense diagonals are handled using CSR_2. (See §4.1 for a discussion of "secondary specializers.")

2.3 OSKI

This method is described in Im et al. [2004]; Demmel et al. [2005]; Vuduc et al. [2002]. The idea is to divide the matrix into (small) dense blocks and perform the multiplication on a block basis, with the code for a single block being unrolled. The goal of this optimization is to increase register reuse. It also reduces the amount of memory required to store indices for the matrix M, since a single index is stored per block. The drawback is that non-zero blocks may still contain zeros, and those have to be added to M, increasing the number of floating-point operations.

For example, suppose n=100, and the chosen block size is 2×2. This divides the matrix into 2500 blocks. Of those, some small percentage will be non-zero — that is, will have at least one non-zero element — and the multiplication code

will iterate over just those blocks, performing a (very efficient) 2×2 multiplication for each. However, many, if not most, of those blocks will contain some zeros, and since the code requires that every block have exactly four elements, those zeros will have to be included. Thus, the efficiency of this method depends on the amount of "fill" added. Because the fill tends to rise with block size, we have never found block sizes greater than 2×2 to give optimal results.

To save space, we omit the code for OSKI, which can be found in Im et al. [2004].

2.4 Unfolding

This method was described in the introduction of this paper. The specialized code consists of exactly n assignment statements, one for each row.

Unfolded code is truly minimal in instruction count, but the code size is proportional to nz. As far as locality in v and w, those will be the same as CSR, since it does the calculations in exactly the same order as CSR does.

Unfolding embeds the matrix values into the code. The alternative is to read them from the *vals* array. If embedded in the code, the compiler puts the unique floating point constants into the data section of the executable file and reuses values when there is duplication. Therefore, the performance of the code may not be exactly the same as the alternative version. We tried both, and experienced that embedding the values in the code gives better performance on average; details are available on our web page [Authors' Web]. The advantage of the alternative version is that it does not require the matrix values to be present; it only requires to know the locations. Hence, code generation can take place at an earlier time in case the matrix shape is available but not the values.

2.5 CSRbyNZ

CSRbyNZ groups the rows of M according to the number of non-zeros they contain (similar to Mellor-Crummey and Garvin [2004]), and generates one loop for each group. The `rows` array 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, but must be reordered to match the order of the loops. So, for example, if there are exactly six rows of M that have three non-zeros, the loop for those rows would be as shown below, where `a` simply indexes over `rows`:

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

In effect, this provides a perfect unrolling of the inner loop of CSR. Moreover, there are rarely very many distinct non-zero counts, so code size is modest; for

example, among our seventeen matrices (see Table 4), the worst case is memplus, with 17,758 rows but only 91 distinct non-zero counts. However, locality in both w and v is likely to be poor; there is no reason to expect locality in v to be any better than it is in CSR, and locality in w is likely to be worse, because the rows handled in a given loop — and thus the elements of w assigned — may be scattered throughout the array. Note that this method requires the elements of M to be reordered, but once this is done, access to them is still sequential.

2.6 Stencil

Like CSRbyNZ, the stencil specializer starts by dividing the rows of M into groups, and then generates a loop for each group. Here, the groups are determined not just by the number of non-zeros in a row, but by the actual pattern of non-zeros, or rather, the pattern of non-zeros *relative to the main diagonal*. Define the *stencil* of row i as the set of numbers $\{j - i \mid M_{i,j} \neq 0\}$; thus, for example, if a row's only non-zeros are on the main diagonal, and just before and after the main diagonal, that row's stencil would be $\{-1, 0, 1\}$. 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 $\{-2, -1, 0, 1, 3\}$, then the loop for this stencil would be:

```
int stencil_2[3] = {2, 4, 6};
for (i=0; i<3; i++) {
   row = stencil_2[i];
   vv = v+row;
   w[row] = vv[-2] * mvals[0] + vv[-1] * mvals[1]
          + vv[0] * mvals[2] + vv[1] * mvals[3]
          + vv[3] * mvals[4];
   mvals += 5;
}
```

The advantage of this method is that it eliminates the `cols` matrix, and as a result, there is no indirect access to `v`. The code size depends on the number of stencils and the amount of computation (that is, the size) of each stencil. In our experiments, code size for this method is usually modest, but there are exceptions. Our worst example is again memplus, which has 16719 distinct stencils, and code size of about 5MB.

This method performs well for banded matrices or matrices that show some regularity with respect to the diagonal.

3 Experimental Setup

The seven target platforms on which we ran our experiments are listed in Table 3. Our programs are written in C and compiled using clang -O3, with two exceptions: MKL and diagonal were compiled using the Intel compiler, icc, be-

Name	Processor & Freq (GHz)	Cache Sizes (Bytes)			Mem	OS	clang	icc
		L1 (I/D)	L2	L3	(GB)			
chicago	Intel Core 2 Duo P8600 @ 2.40	64K	3M	–	2	MacOS Lion 10.7.4	3.2	12.1.5
i2pc4	Intel Xeon L7555 @ 1.87	256K	2M	24M	64	Scientific Linux 6.3	3.2	12.1.3
loome1	Intel Xeon E5640 @ 2.67	128K	1M	12M	12	Linux CentOS 5.8	3.2	–
loome2	Intel Core i7 880 @ 3.07	128K	1M	8M	8	Linux CentOS 5.8	3.2	12.1.4
loome3	Intel Core i5 2400 @ 3.10	32K	256K	6M	8	Linux CentOS 5.8	3.2	12.1.4
pl	Intel Xeon E5405 @ 2.00	32K	6M	–	2	Ubuntu Linux 10.04	3.1	12.0.4
turing	Intel Xeon E5-2620 @ 2.00	32K	256K	15M	16	Ubuntu Linux 12.04	3.2	13.0.0

Table 3: Specification of experimental machines.

Matrix	n	nz	Diagonals		Row	Stencils
			Percentage	Fill rate	nz #	
add32	4960	19,848	25%	0%	6	3,941
cavity02	317	5,923	0%	0%	21	187
cavity23	4,562	131,735	0%	0%	26	440
fidap01	216	4,339	4%	19%	24	216
fidap005	27	279	80%	7%	4	18
fidap031	3,909	91,165	3.5%	19%	28	642
fidap037	3,565	67,591	5%	0%	25	1,716
memplus	17,758	99,147	18%	0%	91	16,719
mhd3200a	3,200	68,026	0%	0%	18	55
mhd4800a	4,800	102,252	0%	0%	17	55
nnc666	666	4,032	0%	0%	11	383
osreg1	2,205	14,133	100%	26%	4	27
pde900	900	4,380	100%	1%	3	9
rdb2048	2,048	12,032	83%	1%	3	18
saylr4	3,564	22,316	100%	9%	5	34
sherman5	3,312	20,793	16%	0%	20	140
utm5940	5,940	83,842	26%	6%	25	176

Table 4: Characteristics of the matrices

cause it produces much better times than clang or gcc for those specializers. For non-vectorized code, clang, gcc, and icc produce similar times.[1]

We show results for 17 matrices from [Matrix Market], listed in Table 4. n and nz are the dimensions and non-zero counts (all matrices are square). "Row nz #" is the number of distinct row non-zero counts; for example, every row in rdb2048 has either 4, 5 or 6 non-zeros. "Stencils" gives the number of stencils. To understand the "Diagonals" column, recall that we use vectorizable code only for diagonals that are at least 80% dense, and that any diagonal handled with this code must have all elements present, even zeros. Under "Percentage," the first column is the percentage of non-zeros handled by diagonal code — that is, the percentage that fall within diagonals that are at least 80% dense;

[1] We were unable to run icc on loome1, so MKL and diagonal were not tested on that machine.

the second column is the percentage of zeros added to those diagonals. For example, utm5940's 21,820 elements are in dense diagonals, and an additional 1279 zeros had to be added to those.

Table 3 and 4 provide some guidance in understanding Table 1. We discuss this further in §4.

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 trials. (We considered using the median of the five times instead of the minimum. We computed the coefficient of variation for both methods, and for both it was very small — always less than 3% and almost always less than 1% — on all machines except loome1. On loome1, the results using minimum were much more stable than those using median. Further details and discussion of this issue can be found on our website [Authors' Web].)

4 Experimental Results

We ran the methods discussed in section 2: CSR_u with u varying from 1 to 10; diagonal; OSKI with block sizes 2×1, 1×2, and 2×2; unfolding; CSR-ByNZ; stencil. We also ran MKL [Intel MKL] (version 10.3) (on all machines except loome1), using function `mkl_dcsrmv`, which performs matrix-vector multiplication on matrices in CSR format; by default, MKL uses multiple processors when available, so to make a fair comparison, we used the `sequential` flag when compiling to force it to use a single processor.

Table 1 gives the "bottom line." Figure 1 gives details for each machine. To reduce clutter, we show only one time for CSR (the best time), and one for OSKI (similarly). Times are normalized to CSR_2, chosen because it is a generally good method that can be employed without latency.

As noted earlier, stencil gives the best time most often, followed by CSR-byNZ, then unfolding, CSR_u for various values of u, and diagonal. In our experiments, neither OSKI nor MKL ever produced the best times.

The obvious question that arises from Figure 1 is "why?" Why do some methods do particularly well for some matrices on some machines? This question immediately brings the possible use of auto-tuning [Püschel et al., 2005; Demmel et al., 2005; Im et al., 2004; Vuduc et al., 2004] to choose the best method for a given matrix on a particular machine (see our discussion in §5). It is clear from Figure 1 that, as in other auto-tuning situations, the answer cannot come simply from looking at the matrix nor from looking at the machine, but must take account of information about both. We do not have a complete answer to the question as yet, but there are some observations that we can make about each method (though almost all admit of some exceptions).

CSR. In 11% of our trials overall, no specialized method could outperform the best unrolling of CSR. (This is particularly true of turing and loome3, where CSR was best on five out of our 17 matrices; in fact, on these

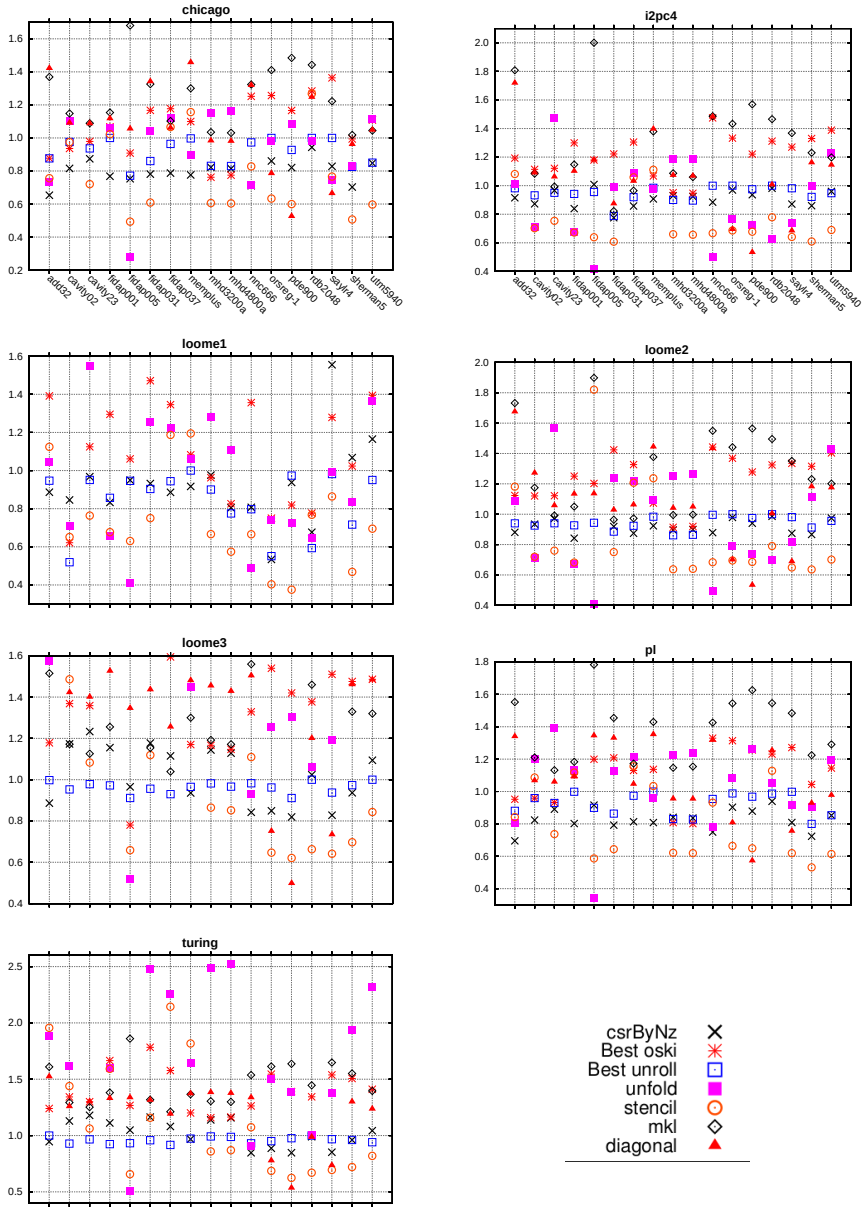

Figure 1: Normalized execution time of the different methods with respect to CSR_2. For unroll and OSKI, the plot shows the time for the best parameter value for each matrix and machine.

machines it was difficult to obtain any speed-ups.) CSR has the one clear advantage that its code is small, but that does not seem sufficient to explain its good performance. In any case, in terms of our library, the question we want to answer is, how can we tell which unrolling of CSR will be best? By looking at the non-zero counts of all the rows (not presented here), we can confirm that the best unrolling is the one in which the most elements are handled in the main loop rather than the clean-up loop.

Diagonal. This is a case where Table 4 almost completely explains the timings. The one matrix for which diagonal is uniformly best is pde900, the most strongly banded matrix: all elements are contained in dense bands, and the dense bands are *very* dense.[2] For orsreg_1 and saylr4, which are also very strongly banded, though not quite as strongly as pde900, diagonal also does well. We note that the properties that make diagonal a good method for a matrix will most likely also favor stencil; see below. We should also again mention that the efficiency of diagonal is limited by our computing with doubles; if we used floats — where vector units can in principle perform four operations at once — or if our machines had a wider vector unit (such as AVX), the results for diagonal might be very different.

OSKI. In our experiments, OSKI rarely performed well. Space keeps us from going into great detail, but to provide one data point as an example: For add32, on every machine, the best version of OSKI is $\text{OSKI}_{2,1}$, and the worst is $\text{OSKI}_{1,2}$, with $\text{OSKI}_{2,2}$ in the middle. The fill ratios are: 41% for both $\text{OSKI}_{1,2}$ and $\text{OSKI}_{2,1}$, 89% for $\text{OSKI}_{2,2}$. With these fill ratios, it is not surprising that OSKI is not competitive; these numbers also show that fill ratio is not the entire story.

Unfolding. Unfolding consistently gives the best times for fidap005, and sometimes does so for cavity02, fidap001, nnc666, and rdb2048. Again, a glance at Table 4 gives at least a partial explanation: These are our smallest matrices (fidap005 is the very smallest), except for pde900, which happens to be handled extremely well by diagonal. However, unfolding is always the best only for fidap005. Otherwise, unfolding shows up mostly on i2pc4, loome1, and loome2; it cannot be a coincidence that these are the machines with the largest L1 (and smallest L2) caches. Likewise, the largest matrix for which unfolding is ever best is rdb2048 (nz = 12,032), and it is best on i2pc4 and loome2, and competitive on loome1.

CSRbyNZ. It is best to contrast CSRbyNZ with stencil. To a first order of approximation, this method is comparatively good when the matrix is too large for unfolding and has too many stencils. For add32, fidap037, and memplus, which have the largest number of stencils, CSRbyNZ is the best on all machines, with a single exception (CSR_4 for fidap037 on loome3).

[2] Reminder: we did not test the diagonal method on loome1.

It also bears mentioning that quite often, when this method is the best, we have gained very little speed-up.

Stencil. Again, the situation is fairly clear: All the matrices do fairly well with this method unless the number of stencils is too large. For the matrices with between 55 and 176 stencils (mdh3200a, mhd4800a, sherman5, and utm5940), stencil is the best on every machine, and for cavity23, with 440 stencils, it is still best on 5 out of 7 machines. On the other hand, stencil count does not tell the whole story: for rdb2048, with only 18 stencils, stencil is the best method only on loome3.

Thus, for some of our matrices, we can predict the best specializer based only on the matrix's characteristics. For others — such as cavity02, fidap001, fidap031, nnc666, and rdb2048 — the best method varies across machines. Moreover, Figure 1 shows that selecting the second-best method can result in a substantial loss of performance relative to the best, sometimes exceeding a factor of 2.

Finally, we were surprised that performance of MKL in general is very poor, even slower than our reference CSR_2. This is probably because MKL is tuned for parallel computation rather than sequential.

4.1 Mixed methods

Mixing methods — partitioning a matrix and using different methods for the parts — can produce additional speed-ups. Although we have not explored this space fully (and therefore do not show these results in Table 1), in a few cases, we have obtained substantial speed-ups in this way. These are listed in Table 5. Those were all cases where stencil was the best performer, and improvements were obtained by mixing stencil with a different "secondary specializer."

There are two basic ideas here: (1) Consider the stencil code shown in §2.6. If a stencil is "unpopular" — there is just one row with that stencil (that is, if, in the example in §2.6, array `stencil_2` had one element) — we dispense with the loop, and obtain code resembling unfolding. But unfolding is not necessarily optimal; we can instead gather all those rows and handle them by a different method. (2) We can limit the amount of stencil code in several ways, of which we mention one: We can partition M into a band of a certain width around the main diagonal, handling the elements within the band by stencil and the others by a different method. The effect is fewer stencils, with higher popularity. (To see this, consider the extreme case of a band of width three; there are only a total of eight possible stencils — in practice, only four, since the main diagonal is usually 100% dense.) We can use both methods: consider stencils only within a band, and take all the elements either outside the band, or inside it but in unpopular stencils, and treat them separately. Table 5 shows the cases where we obtained significant speed-ups; reported speed-up is relative to the best time, as given in Table 1; bandwidth of ∞ means method (2) was not employed.

Machine	Matrix	Band	Secondary	Speed-up
chicago	cavity02	10	CSRbyNZ	10%
	nnc666	50	CSRbyNZ	17%
loome1	cavity02	20	CSRbyNZ	18%
	fidap001	10	CSRbyNZ	32%
	rdb2048	100	CSR_2	24%
	saylr4	∞	unfolding	27%
loome3	nnc666	∞	CSRbyNZ	14%
pl	nnc666	∞	CSRbyNZ	15%
turing	fidap005	50	CSRbyNZ	22%
	nnc666	∞	CSRbyNZ	17%

Table 5: Speed-ups from mixed methods, relative to the best times, as given in Table 1.

With all the possible band widths and choices of secondary specializers, the "design space" here is obviously enormous. We plan to study the possibilities further.

Lastly, we mention a specializer whose use inherently involves partitioning the matrix: diagonal. Since the vectorizable code handles only values that occur in dense diagonals, a secondary method is used for the remainder. In our experiments reported above, we used CSR_2. We tried other secondary specializers, but a glance at the previous results (especially Table 4) shows why this effort was bound to fail: The only matrices for which diagonal works well are those where virtually every element is in a dense diagonal, but then there are few or no elements left for the secondary specializer.

5 Applications

Specialization is useful when a matrix is going to be multiplied by many vectors. There are frequently cases where there are many matrices that have the exact same sparseness structure (in Matrix Market these are called "pattern matrices"); since all our methods except unfolding rely only on the sparseness structure, these methods could be used to optimize many matrix multiplications.

Ultimately, one would prefer to have a simple matrix library that could be called as simply as any other, but would choose, and generate, the best code at run time. The library would be given a sparse matrix `M` of doubles, represented in compressed-sparse-row (CSR) format, and the library would produce a pointer to a function of type `void multByM (double v[], double w[])`. When called subsequently, `multByM` multiplies `M` by `v` and places the result in `w`. (The OSKI library [Im et al., 2004; Demmel et al., 2005; Vuduc et al., 2002] operates similarly.)

When first presented with `M`, the system determines which method will produce the most efficient `multByM`. It may determine that a certain method for which code already exists is the best, and immediately returns a pointer to the

existing function; or it may determine that a specialized code which must be generated at run-time, will be most efficient. This process itself will take time, and run-time generation of the specialized code, if that is the decision, will take even more; in any case, the system cannot produce overall speed-ups if the matrix is to be multiplied only a small number of times.

This library organization raises several questions:

1. What methods of generating `multByM` are likely to produce efficient code and what are the kind of speedups that these methods can deliver? This is the question we address in this paper.

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) is a difficult one because the best method varies according to the matrix and machine. The solution is auto-tuning [Whaley et al., 2001; Frigo, 1999; Püschel et al., 2005; Li et al., 2005; Demmel et al., 2005; Datta et al., 2008; Vuduc et al., 2004]. Here, one gathers information about the machine at "install time," by generating codes for several matrices and timing them. This information is fed into the run-time specialization process, which uses it, together with characteristics of the matrix `M`, to determine how to generate `multByM`. We do not yet know how difficult this will be, as this is ongoing research. However, we discussed the general ideas in §4.

To address question (3), auto-tuning (and, if needed, run-time specialization) can be performed in parallel, while the library uses the generic CSR method (or another existing code) until the specialized code is ready. For fast generation of code, template-based approach can be taken [Smith et al., 2003; Consel et al., 2004], where pieces of binary code that contain holes are filled in with constants and composed to build executable code.

The work presented here discusses generating sequential code for a problem that is highly parallelizable. To see how optimizing sequential execution can help, consider a scenario where one decomposes a large matrix into multiple smaller ones, by taking various load-balancing factors into account, and assigns each to a thread. The threads will run sequential code to compute their contribution to the overall result. The findings reported here can help in the process of deciding which format and method to use for which submatrix.

6 Related Work

Sparse matrix-dense vector multiplication is an operation that is used in many scientific problems. Using a blocking order to improve locality has been studied in the OSKI project [Im et al., 2004]. There are several approaches that utilize the structure of the matrix to improve performance, two of which are

Mellor-Crummey and Garvin [2004] and Shantharam et al. [2011]. A number of researchers have looked at multi-core implementations [D'Azevedo et al., 2005; Jain, 2008; Buluc et al., 2011].

The library described in §5 will use "auto-tuning," whereby relevant machine characteristics are gathered at "install time," and combined with characteristics of the input data to determine the best method for a problem. 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 Whaley et al. [2001]; Frigo [1999]; Püschel et al. [2005]; Li et al. [2005]; Datta et al. [2008]; Vuduc et al. [2004].

Run-time code generation is used routinely in "just-in-time" compilers. For example, Google's V8 compiler for JavaScript [V8] overcomes the inherent inefficiency of dynamic typing by specializing the code to the data types that occur most often at run time. However, in this application of run-time code generation, the programmer has little or no control.

In program specialization — also called *code generation*, *partial evaluation*, or *staging* — the programmer determines exactly what code will be generated. The area has been heavily studied, especially with respect to language features, such as type-checking, that promote simplicity and safety of specialization [Taha and Nielsen, 2003; Westbrook et al., 2010; Choi et al., 2011]. Work in this area specifically addressing high-performance for realistic applications includes work on marshalling [Aktemur et al., 2005; Cohen and Herrmann, 2005] and on code-optimizing transformations [Cohen et al., 2006]. With *run-time* specialization, the focus moves toward the efficiency of specialization itself [Kamin et al., 2003; Smith et al., 2003].

Our work draws from these three areas: We use *(run-time) program specialization* to optimize *sparse matrix-vector multiplication*, taking into account the need for *auto-tuning*, since the most appropriate method varies according to the machine and matrix.

7 Future work and conclusions

We have shown that the method of program specialization can be used to obtain speed-ups for sparse matrix-dense vector multiplication, in those cases where one matrix is to be multiplied by many vectors. The obvious method of completely unfolding the computation is sometimes effective, but for larger matrices, more subtle methods are needed. We have provided an extensive benchmark to compare the performance of methods. We have also shown that there is no one best method, and we discussed the use of "auto-tuning" in a proposed library for this problem. Lastly, we showed how partitioning a matrix and using different methods for different parts can be advantageous; the design space becomes very large here; we do not yet know how to find the *very best* multiplier for a given matrix.

Our goal in this work was to address the question "How much speedup can be obtained by program specialization for sparse matrix-vector multiplication?"

We have ambitions beyond this problem. Obviously, there are many other matrix calculations that might be amenable to specialization. Not only the calculations where the input data can be divided into an early, stable part and a late, dynamic part, but also the problems where a large number of variations exist can benefit greatly from the generative approach combined with auto-tuning. The 19^{th} Nii Shonan meeting [Kiselyov et al., 2012; Aktemur et al., 2013] has compiled a list of such problems including a staged matrix algebra library, a dynamic programming library, a staged hidden Markov model, and an image processing pipeline, among others.

References

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

B. Aktemur, Y. Kameyama, O. Kiselyov, and C. Shan. Shonan challenge for generative programming: short position paper. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, PEPM '13, pages 147–154, New York, NY, USA, 2013. ACM.

Authors' Web. URL `removed-for-blind-review.` Accessed January 2013.

A. Buluc, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. *IPDPS '11*, 13: 721–733, 2011.

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.

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

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.

C. Consel, J. Lawall, and A. Le Meur. A tour of Tempo: a program specializer for the C language. *Sci. Comput. Program.*, 52(1-3):341–370, 2004.

K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08*, pages 4:1–4:12, 2008.

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

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

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.

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

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.

Intel MKL. URL `http://software.intel.com/en-us/intel-mkl`. Accessed January 2013.

A. Jain. poski: An extensible autotuning framework to perform optimized spmvs on multicore architectures. Master's thesis, University of California at Berkeley, 2008.

N. Jones, C. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

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

O. Kiselyov, C. Shan, and Y. Kameyama. Nii Shonan Meeting 19, 2012. `http://www.nii.ac.jp/shonan/seminar019/` and `http://www.nii.ac.jp/shonan/wp-content/uploads/2011/09/No.2012-4.pdf`.

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

Matrix Market. URL `http://math.nist.gov/MatrixMarket`. Accessed January 2013.

J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18 (2):225–236, May 2004.

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.

Manu Shantharam, Anirban Chatterjee, and Padma Raghavan. Exploiting dense substructures for fast sparse matrix vector multiplication. *Int. J. High Perform. Comput. Appl.*, 25(3):328–341, August 2011.

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.

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

W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

V8. V8 JavaScript Engine. URL `http://code.google.com/p/v8/`. Accessed January 2013.

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.

R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, February 2004.

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.

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