Optimization by Run-time Specialization for Sparse Matrix-Vector Multiplication

(Submitted for publication)

Sam Kamin
Univ. of Illinois
kamin@illinois.edu

Baris Aktemur
Ozyegin Univ.
baris.aktemur@ozyegin.edu.tr

Maria Garzaran
Univ. of Illinois
garzaran@illinois.edu

Edgar Black
Univ. of Illinois
efblack2@illinois.edu

Asim Yildiz
Ozyegin Univ.
asim.yildiz@ozu.edu.tr

Abstract

Run-time specialization is the process of generating programs based on information available only at run time. This technique has the potential of generating highly efficient codes, at the expense of the overheads of the run-time code generation. It is applicable when some input data is used repeatedly while other input data varies.

In this paper we explore the potential for obtaining speed-ups for sparse matrix-dense vector multiplication using run-time specialization, in the case where a single matrix is to be multiplied by many vectors. We experiment with three methods involving run-time 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 run-time 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.

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 $\text{multBy}_M(v, w)$ 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; a popular technique of automatic specialization is called partial evaluation. 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 $v$, $\text{multBy}_M$ consists of a long sequence of assignment statements of the form

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

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 [12]). This code is “optimal” in the sense of producing the minimum instruction count; a standard Compressed-Sparse-Row (CSR) loop (see section 3.1) will execute per-

[Copyright notice will appear here once ‘preprint’ option is removed.]
haps 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 3) 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 \( \text{CSR}_u \).

**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.)

**OSKI.** This is the method implemented in the OSKI library [14, 17, 25] 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. 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.

Notice that not all of our methods involve run-time 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 run-time generation of code (OSKI, diagonal); and those that require run-time code generation (unfolding, CSRbyNZ, and stencil).

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 \( \text{CSR}_2 \) (chosen as the most efficient non-specialized method). This table illustrates the two main points of this paper:

1. Speed-ups can be obtained by run-time specialization. In most cases, one of the methods involving run-time 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 \((17 \times 7)\) trials, the best specializers were: stencil (56 times), CSRbyNZ (29), unfolding (15), diagonal (7), \( \text{CSR}_3 \) (7), \( \text{CSR}_4 \) (2), \( \text{CSR}_5 \) (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 section 5.1).

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 [5]. (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.) To our knowledge, one of the proposed methods — CSRbyNZ — is new.

Our experimental results show that methods that require run-time 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. In this work, we are not considering the runtime cost of specialization; rather, the question that we are first trying to answer is whether run-time specialization can produce speedups, and if so, of what magnitude. Of course, the time devoted to run-time specialization is a major issue which we will study in future work; we outline our efforts in section 2.

Our experiments include comparisons with the running times of the Intel math library MKL [1]. 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 Section 2 we explain the goals of the project, and the code library that we envision building. At its heart is the set of methods listed above for performing matrix-vector multiplication; in Section 3, we explain these methods in detail; Section 4 describes our experimental setup; Section 5 shows our performance numbers. Section 6 discusses related work. Finally, future work and conclusions are presented in Section 7.

2. A library based on run-time specialization

The work presented here is the first step in the creation of a library for matrix-vector multiplication that will use run-time specialization, auto-tuning, and machine learning techniques to predict the best method. 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 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 [11, 14, 15, 21, 22, 27]. 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 do have some general ideas that we discuss in section 5.

To minimize latency, question 3, we are developing a fast run-time C compiler based on clang [4], using techniques similar to the ones developed for previous systems, such as Jumbo [19].

We are not currently generating parallel code. In general, our methods seem to be easily parallelizable, but this is left for future work. We plan to make use of parallelism in our library in another way, namely to perform auto-tuning and

<table>
<thead>
<tr>
<th></th>
<th>chicag0</th>
<th>i2pc4</th>
<th>loome1</th>
<th>loome2</th>
<th>loome3</th>
<th>pl</th>
<th>turing</th>
</tr>
</thead>
<tbody>
<tr>
<td>add32</td>
<td>CSRbyNZ(65)</td>
<td>CSRbyNZ(91)</td>
<td>CSRbyNZ(88)</td>
<td>CSRbyNZ(88)</td>
<td>CSRbyNZ(88)</td>
<td>CSRbyNZ(89)</td>
<td>CSRbyNZ(94)</td>
</tr>
<tr>
<td>cavity02</td>
<td>CSRbyNZ(81)</td>
<td>stencil(70)</td>
<td>CSR, (51)</td>
<td>unfolding(71)</td>
<td>CSR, (95)</td>
<td>CSRbyNZ(82)</td>
<td>CSR, (92)</td>
</tr>
<tr>
<td>cavity23</td>
<td>stencil(72)</td>
<td>stencil(75)</td>
<td>stencil(76)</td>
<td>stencil(76)</td>
<td>stencil(97)</td>
<td>stencil(73)</td>
<td>CSRbyNZ(96)</td>
</tr>
<tr>
<td>fidap001</td>
<td>CSRbyNZ(76)</td>
<td>stencil(67)</td>
<td>unfolding(65)</td>
<td>unfolding(67)</td>
<td>CSR, (97)</td>
<td>CSRbyNZ(80)</td>
<td>CSRbyNZ(92)</td>
</tr>
<tr>
<td>fidap005</td>
<td>unfolding(27)</td>
<td>unfolding(41)</td>
<td>unfolding(41)</td>
<td>unfolding(41)</td>
<td>unfolding(51)</td>
<td>unfolding(34)</td>
<td>unfolding(50)</td>
</tr>
<tr>
<td>fidap031</td>
<td>stencil(60)</td>
<td>stencil(60)</td>
<td>stencil(75)</td>
<td>stencil(75)</td>
<td>CSR, (95)</td>
<td>stencil(64)</td>
<td>CSRbyNZ(95)</td>
</tr>
<tr>
<td>fidap037</td>
<td>CSRbyNZ(78)</td>
<td>CSRbyNZ(85)</td>
<td>CSRbyNZ(88)</td>
<td>CSRbyNZ(87)</td>
<td>CSR, (92)</td>
<td>CSRbyNZ(81)</td>
<td>CSR, (91)</td>
</tr>
<tr>
<td>memplus</td>
<td>CSRbyNZ(77)</td>
<td>CSRbyNZ(90)</td>
<td>CSRbyNZ(91)</td>
<td>CSRbyNZ(92)</td>
<td>CSRbyNZ(93)</td>
<td>CSRbyNZ(80)</td>
<td>CSRbyNZ(96)</td>
</tr>
<tr>
<td>mhd3200a</td>
<td>stencil(60)</td>
<td>stencil(66)</td>
<td>stencil(66)</td>
<td>stencil(63)</td>
<td>stencil(86)</td>
<td>stencil(62)</td>
<td>stencil(85)</td>
</tr>
<tr>
<td>mhd4800a</td>
<td>stencil(60)</td>
<td>stencil(65)</td>
<td>stencil(57)</td>
<td>stencil(63)</td>
<td>stencil(85)</td>
<td>stencil(61)</td>
<td>stencil(87)</td>
</tr>
<tr>
<td>nnc666</td>
<td>CSRbyNZ(71)</td>
<td>unfolding(50)</td>
<td>unfolding(48)</td>
<td>unfolding(49)</td>
<td>CSRbyNZ(84)</td>
<td>CSRbyNZ(74)</td>
<td>CSRbyNZ(84)</td>
</tr>
<tr>
<td>orsreg1</td>
<td>stencil(63)</td>
<td>stencil(68)</td>
<td>stencil(40)</td>
<td>stencil(69)</td>
<td>stencil(64)</td>
<td>stencil(66)</td>
<td>stencil(68)</td>
</tr>
<tr>
<td>pde900</td>
<td>diagonal(52)</td>
<td>diagonal(53)</td>
<td>stencil(37)</td>
<td>diagonal(53)</td>
<td>diagonal(49)</td>
<td>diagonal(57)</td>
<td>diagonal(53)</td>
</tr>
<tr>
<td>rdb2048</td>
<td>CSRbyNZ(94)</td>
<td>unfolding(62)</td>
<td>CSR, (39)</td>
<td>unfolding(69)</td>
<td>stencil(66)</td>
<td>CSRbyNZ(93)</td>
<td>stencil(67)</td>
</tr>
<tr>
<td>sayh4</td>
<td>diagonal(66)</td>
<td>stencil(64)</td>
<td>stencil(86)</td>
<td>stencil(64)</td>
<td>stencil(64)</td>
<td>stencil(61)</td>
<td>stencil(69)</td>
</tr>
<tr>
<td>sherman5</td>
<td>stencil(50)</td>
<td>stencil(60)</td>
<td>stencil(46)</td>
<td>stencil(63)</td>
<td>stencil(69)</td>
<td>stencil(53)</td>
<td>stencil(72)</td>
</tr>
<tr>
<td>utm5940</td>
<td>stencil(59)</td>
<td>stencil(68)</td>
<td>stencil(69)</td>
<td>stencil(70)</td>
<td>stencil(84)</td>
<td>stencil(61)</td>
<td>stencil(81)</td>
</tr>
</tbody>
</table>

Table 1. Best method for all matrices/machines; run time as a percentage of CSR. Characteristics of the machines and matrices can be found in Tables 3 and 4.
specialization; this will reduce the latency that is the major drawback of using run-time specialization.

3. 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 section 5.1.

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 × 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 [3]. The Python script that generates the code is available there as well.)

3.1 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; for each 0 ≤ i ≤ n − 1, rows[i] gives the index in mvalues and columns at which the non-zeros for row i begin. Actually, rows has length n + 1, with rows[n] containing 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. CSRn is CSR with an unrolling factor of u; “CSR” means CSR1. If u > 1, the code consists of an unrolled loop handling u elements at a time and a “cleanup” CSR1 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 $v$.

### 3.2 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 CSR2. (See section 5.1 for a discussion of “secondary specializers.”)

### 3.3 OSKI

This method is described in [14, 17, 25]. 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 $2 \times 2$. 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) $2 \times 2$ 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 $2 \times 2$ to give optimal results.

To save space, we omit the code for OSKI, which can be found in [17].

### 3.4 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.

### 3.5 CSRbyNZ

CSRbyNZ 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, 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 $rows$:

```c
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.

---

Table 2. Some characteristics of the methods

<table>
<thead>
<tr>
<th></th>
<th>CSR</th>
<th>unrolled CSR</th>
<th>Diagonal</th>
<th>OSKI</th>
<th>Unfolding</th>
<th>CSRbyNZ</th>
<th>Stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size</td>
<td>small</td>
<td>small</td>
<td>small</td>
<td>large</td>
<td>Modest</td>
<td>Modest</td>
<td>Modest</td>
</tr>
<tr>
<td></td>
<td>(proportional to degree of unroll)</td>
<td>(proportional to block size)</td>
<td>(proportional to # of nonzeros)</td>
<td>(proportional to count of nonzeros)</td>
<td>(proportional to # of stencils)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect Access to v?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes, but fewer than CSR</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Zero Filling?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

2012/10/2
3.6 Stencil

Like CSRbyNZ, the stencil specialist 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:

```c
int stencil_2[3] = {2, 4, 6};
for (i=0; i<3; i++) {
    row = stencil_2[i];
    vv = v[row];
    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.

4. 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, because it produces much better times than clang or gcc for those specialize.

Table 3. Specification of experimental machines.

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

Table 4. Characteristics of the matrices

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nz #</th>
<th>Diag. Fr</th>
<th>Row nz #</th>
<th>Stencils</th>
</tr>
</thead>
<tbody>
<tr>
<td>add32</td>
<td>4960</td>
<td>23,884</td>
<td>25% 0%</td>
<td>6</td>
<td>3,941</td>
</tr>
<tr>
<td>cavity02</td>
<td>317</td>
<td>7,327</td>
<td>0% 0%</td>
<td>21</td>
<td>187</td>
</tr>
<tr>
<td>cavity23</td>
<td>4,562</td>
<td>138,187</td>
<td>0% 0%</td>
<td>26</td>
<td>440</td>
</tr>
<tr>
<td>fidap01</td>
<td>216</td>
<td>4,374</td>
<td>4% 19%</td>
<td>24</td>
<td>216</td>
</tr>
<tr>
<td>fidap005</td>
<td>27</td>
<td>279</td>
<td>80% 7%</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>fidap031</td>
<td>3,909</td>
<td>115,299</td>
<td>3% 19%</td>
<td>28</td>
<td>642</td>
</tr>
<tr>
<td>fidap037</td>
<td>3,565</td>
<td>67,591</td>
<td>5% 0%</td>
<td>25</td>
<td>1,716</td>
</tr>
<tr>
<td>memplus</td>
<td>17,758</td>
<td>126,150</td>
<td>18% 0%</td>
<td>91</td>
<td>16,719</td>
</tr>
<tr>
<td>mhd3200a</td>
<td>3,200</td>
<td>68,026</td>
<td>0% 0%</td>
<td>18</td>
<td>55</td>
</tr>
<tr>
<td>mhd4800a</td>
<td>4,800</td>
<td>102,252</td>
<td>0% 0%</td>
<td>17</td>
<td>55</td>
</tr>
<tr>
<td>mnc666</td>
<td>666</td>
<td>4,044</td>
<td>0% 0%</td>
<td>11</td>
<td>383</td>
</tr>
<tr>
<td>osreg1</td>
<td>2,205</td>
<td>14,133</td>
<td>100% 26%</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>pde900</td>
<td>900</td>
<td>4,380</td>
<td>100% 1%</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>rdb2048</td>
<td>2,048</td>
<td>12,032</td>
<td>83% 1%</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>saylr4</td>
<td>3,564</td>
<td>22,316</td>
<td>100% 9%</td>
<td>5</td>
<td>34</td>
</tr>
<tr>
<td>sherman5</td>
<td>3,312</td>
<td>20,793</td>
<td>16% 0%</td>
<td>20</td>
<td>140</td>
</tr>
<tr>
<td>utm5940</td>
<td>5,940</td>
<td>83,842</td>
<td>26% 6%</td>
<td>25</td>
<td>176</td>
</tr>
</tbody>
</table>

Table 4. Characteristics of the matrices

We show results for 17 matrices from the Matrix Market [5], 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 “Diag” 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 “Diag,” 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; 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 section 5.

1 We were unable to run icc on loome1, so mkl and diagonal were not tested on that machine; the final version of the paper will include those timings.
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 [3].)

5. Experimental Results

We ran the methods discussed in section 3: $CSR_u$ with $u$ varying from 1 to 10; diagonal; OSKI with block sizes $2 \times 1$, $1 \times 2$, and $2 \times 2$; unfolding; CSRbyNZ; stencil. We also ran MKL [1] (version 10.3) (on all machines except loome1), using function $\text{mkl}_1 \_\text{dcsrmv}$, 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 CSRbyNZ, 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? Aside from the inherent interest of the question, it has a direct impact on our ambition to build the library described in section 2. The first job of the library code is auto-tuning: choosing the best method for the given matrix on that machine. 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 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. (On the other hand, the library would not really need to make this determination; since any unrolling of CSR can be used without latency, we could simply run various unrollings and time them until we settle on the best one.)

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. For orsreg1 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 OSKI$_{2,1}$, and the worst is OSKI$_{1,2}$, with OSKI$_{2,2}$ in the middle. The fill ratios are: 41% for both OSKI$_{1,2}$ and OSKI$_{2,1}$, 89% for 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, nnce666, 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
(CSR₂ for fidap037 on loome3). 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 35 and 176 stencils (mdn3200a, mbd4800a, 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 CSR₂.

5.1 Mixed methods

The “auto-tuning” analysis of the previous section is useful for another purpose: it can tell us how we might mix methods to produce better speed-ups. By “mixing methods,” we mean partitioning a matrix and using different methods for the parts.

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 section 3.6. If a stencil is “unpopular” — there is just one row with that stencil (that is, if, in the example in section 3.6, array s:stencil1_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 $\infty$ means method (2) was not employed.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Matrix</th>
<th>Band</th>
<th>Secondary</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>chicago</td>
<td>cavity02</td>
<td>10</td>
<td>CSRbyNZ</td>
<td>10%</td>
</tr>
<tr>
<td>nnc666</td>
<td>50</td>
<td>CSRbyNZ</td>
<td>17%</td>
<td></td>
</tr>
<tr>
<td>loome1</td>
<td>cavity02</td>
<td>20</td>
<td>CSRbyNZ</td>
<td>18%</td>
</tr>
<tr>
<td>fidap001</td>
<td>10</td>
<td>CSRbyNZ</td>
<td>32%</td>
<td></td>
</tr>
<tr>
<td>rdb2048</td>
<td>100</td>
<td>CSR₂</td>
<td>24%</td>
<td></td>
</tr>
<tr>
<td>saylr4</td>
<td>$\infty$</td>
<td>unfolding</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>loome3</td>
<td>nnc666</td>
<td>$\infty$</td>
<td>CSRbyNZ</td>
<td>14%</td>
</tr>
<tr>
<td>pl</td>
<td>nnc666</td>
<td>$\infty$</td>
<td>CSRbyNZ</td>
<td>15%</td>
</tr>
<tr>
<td>turing</td>
<td>fidap005</td>
<td>50</td>
<td>CSRbyNZ</td>
<td>22%</td>
</tr>
<tr>
<td>nnc666</td>
<td>$\infty$</td>
<td>CSRbyNZ</td>
<td>17%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Speed-ups from mixed methods

6. 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 [17]. A number of researchers have looked at multi-core implementations [7, 13, 18].

The library described in section 2 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 [11, 15, 21, 22, 27].

Run-time code generation is used routinely, in “just-in-time” compilers. For example, Google’s V8 compiler for JavaScript [2] 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 not 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 [8, 24, 26]. Work in this area specifically addressing high-performance for realistic applications includes work on marshalling [6, 9] and on code-optimizing transformations [10]. With run-time specialization, the focus moves toward the efficiency of specialization itself [19, 23].

Our work draws from these three areas: We use run-time specialization to optimize 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 run-time 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 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. We outlined the working of this library. It leaves several research questions:

• How to auto-tune, as outlined in section 2.
• How to optimize code generation [19]. We are developing an LLVM-based run-time compiler for a subset of C.
• How to generate optimized code. LLVM is designed to allow for pluggable optimization passes. The issue we are interested in is the special problem of optimizing generated code. As has been observed [16], generated code has somewhat different characteristics than handwritten code (most obviously, it is often much longer), and compilers are tuned for the latter.
• Calculating stencils is a complex job, and is important because the stencil method is so often the best performer. We need to find a fast algorithm for this problem.
• Can we find still more efficient methods? For the two largest matrices, cavity23 and memplus, we have failed to achieve substantial speed-ups on several machines.

We have ambitions beyond matrix-vector multiplication. Obviously, there are many other matrix calculations that might be amenable to run-time 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 19th Nii Shonan meeting [20] 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


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.