Autotuning Runtime Specialization for Sparse Matrix-Vector Multiplication

BUSE YILMAZ and BARIŞ AKTEMUR, Ozyegin University
MARÍA J. GARZARÁN, University of Illinois at Urbana-Champaign and Intel Corporation
SAM KAMIN, University of Illinois at Urbana-Champaign and Google
FURKAN KIRAÇ, Ozyegin University

Runtime specialization is used for optimizing programs based on partial information available only at runtime. In this paper we apply autotuning on runtime specialization of Sparse Matrix-Vector Multiplication to predict a best specialization method among several. In 91% to 96% of the predictions, either the best or the second-best method is chosen. Predictions achieve average speedups that are very close to the speedups achievable when only the best methods are used. By using an efficient code generator and a carefully designed set of matrix features, we show the runtime costs can be amortized to bring performance benefits for many real-world cases.

1. INTRODUCTION

Sparse Matrix Vector Multiplication (SpMV) is the kernel operation used in many iterative methods to solve large linear systems of equations. Sparse matrices appear in many problem domains. In the scientific or engineering domain they are obtained by discretization of partial differential equations and represent physical phenomena,

such as heat, electrodynamics, or quantum mechanics. They can also be obtained from graphs, in which case they represent the Internet structure or social interactions. For iterative solutions, various approaches, such as Krylov subspace methods, can be used. Usually, they converge after a large number of iterations. Thus, they are often combined with preconditioners to decrease the number of iterations. Preconditioning can increase the running time of each iteration, but the total runtime is reduced. The problem with preconditioning is that finding a good preconditioner is usually viewed as a combination of art and science [Saad 2003]. For some matrices, there is no good preconditioner. Thus, the problem of generating efficient code for SpMV, the kernel operation in these iterative solvers, is a critical problem; it has been and continues to be extensively researched [D'Azevedo et al. 2005; Jain 2008; Buluç et al. 2009, 2011; Kourtis et al. 2011; Williams et al. 2009; Belgin et al. 2011; Mellor-Crummey and Garvin 2004; Bell and Garland 2009; Venkat et al. 2015; Liu et al. 2013].

In a previous work [Kamin et al. 2014] we investigated how much speedup can be obtained by applying runtime specialization for the SpMV operation ($w \leftarrow w + Av$). In that work, we experimented with five methods requiring specialization and compared them with methods that did not, including Intel® Math Kernel Library (Intel MKL) [MKL 2013], and other state-of-the-art libraries such as CSB [Buluç et al. 2009], BiCSB [Buluç et al. 2011], and CSX [Kourtis et al. 2011], whose code was available online. We found that, in most cases, a method using runtime specialization was the fastest. However, we also found that no single method is the best, as the best method varies across machines and across matrices. While offline code generation is possible for many problem domains (e.g. when the matrix, or at least its pattern, is known beforehand), in the general case, runtime specialization can only be profitable if:

—the best specialization method can be predicted without having to generate and run all the code variants;
—code of the method predicted to be the best can be generated quickly.

In this paper, we address the issues above and show that runtime specialization of SpMV for real-world matrices is feasible. Our **contributions** are threefold:

—We investigate how accurately we can predict the best SpMV method for a given matrix. We use a Support Vector Machine (SVM) machine-learning technique to predict the best among six methods (including Intel MKL as the baseline method)
—We give a list of matrix features to determine the performance of SpMV. Several of these features are unique to our work. We also experiment with an early-exit strategy when extracting the matrix features to decrease matrix analysis costs significantly.
—We developed an end-to-end special-purpose compiler that takes a matrix and generates specialized executable code for the X86_64 architecture at runtime. We show that the runtime costs and break-even points are low enough that runtime specialization of SpMV for many real-world matrices is feasible.

The novelty of our work is not in the use of autotuning for SpMV; that problem has been studied extensively, in particular for selecting a matrix storage format (see Section 8 for related work). We also do not claim that we generate aggressively optimized SpMV code, for which there also exist outstanding body of work. The novelty of our work lies in using autotuning for selecting a runtime specialization method, defining the matrix features for this purpose and in generating long SpMV code very rapidly. These make runtime specialization of SpMV profitable in practice.

Organization: In Section 2, we explain the SpMV specialization methods we evaluated. Section 3 presents how we do code generation. Section 4 describes the autotuning approach we applied. Our experimental setup and results are presented in Sections 5 and 6, respectively. In Section 7, we evaluate the latency incurred by runtime specialization. Section 8 gives related work. Finally, Section 9 presents our conclusions.

2. BACKGROUND: SPECIALIZATION METHODS CONSIDERED

In this section, we briefly describe the methods that we use to specialize the SpMV code. For performance comparison, we use Intel MKL's SpMV as the baseline implementation. We could not use the AMD Core Math Library (ACML) [ACML 2013] on our testbed machine that has an AMD CPU, because ACML lacks SpMV.

In the discussion of the methods below, we assume A is an N×N matrix, with NZ nonzeros. In the code snippets, the `rows` array contains the row indices, the `cols` array contains the column indices, and the `vals` array contains the nonzero elements of the matrix; v is the input vector, w is the output vector. The type of `rows` and `cols` is `int*`; the type of `vals`, v, and w is `double*`.

We feed matrices into Intel MKL in *Compressed Sparse Row* (CSR) format. In this format, the `vals` array contains NZ double precision floating-point values; the `cols` array contains the column indices of nonzero elements (NZ integers); the `rows` array contains, for each row, the starting/ending index of elements in the `cols` and `vals` arrays (N+1 integers). Hence, the data size is (NZ+N+1)×4 + NZ×8 bytes, assuming 4-byte integers. The interpretation and size of the arrays change according to the method.

CSRbyNZ

This method groups the rows of A according to the number of nonzeros they contain (i.e., the *row length*) and generates a loop for each group of rows [Mellor-Crummey and Garvin 2004]. This method gains its efficiency from long basic blocks in each loop, which can be compiled efficiently. It provides, in effect, a perfect unrolling of the inner loop of CSR and so reduces loop overhead, which is important in SpMV [Goumas et al. 2008]. CSRbyNZ would generate the following code for 100 rows with a length of 3:

```c
for (int a = 0, b = 0; a < 100; a++, b += 3) {
  int row = rows[a];
  w[row] += vals[b]*v[cols[b]] + vals[b+1]*v[cols[b+1]] + vals[b+2]*v[cols[b+2]];
}
rows += 100;  cols += 100*3;  vals += 100*3; // Set the pointers for the next loop.
```

Data order: CSRbyNZ reorders the matrix data to group rows with the same length together. Because of reordering, accesses to the output vector w are not sequential.

Data size: The `rows` array contains the indices of nonempty rows. Hence, the data size of the matrix is the same as CSR, except for when there are rows with no elements.

Code size: Since this method generates one for-loop for each row length, and the body of a loop contains as many multiplications as the row length, the code size is proportional to the number of distinct row lengths and their sum.

RowPattern

This method analyzes the matrix to find the exact pattern of nonzero entries in each row of A and generates, for each pattern, a loop that handles all the rows that have that pattern. Specifically, the pattern of each row is defined as the location of the nonzeros with respect to the main diagonal. So if row r has nonzeros in columns $r - 2, r, r + 1$, and $r + 3$, then its pattern would be $\{-2, 0, 1, 3\}$. Sample code corresponding to this row pattern, assuming there are 100 rows with that pattern, is given below.

```c
for (int a = 0, b = 0; a < 100; a++, b += 4) {
  int row = rows[a];
  w[row] += vals[b]   * v[row-2] + vals[b+1] * v[row]
          + vals[b+2] * v[row+1] + vals[b+3] * v[row+3];
}
rows += 100;  vals += 100*4; // Set the pointers for the next loop.
```

Data order: RowPattern reorders the matrix data to group rows with the same pattern together; similarly to CSRbyNZ, accesses to the output vector w are not sequential.

Data size: RowPattern provides matrix data reduction by making the column indices explicit in the code and thus eliminating the need to store column indices. This is a saving of NZ-many integer values. Similarly to CSRbyNZ, the length of the `rows` array is equal to the number of nonempty rows.

Code size: For matrices with a modest number of row patterns, this method can be the most efficient. However, if there are many patterns, the code can get quite large, reducing its efficiency. Since this method generates one for-loop for each row pattern, and the body of a loop contains as many multiplications as the length of the pattern, the code size is proportional to the number of row patterns and the sum of their lengths.

RowPattern turns indirect indexing on the vector v (e.g., `v[cols[b]]`) to direct indexing (e.g., `v[row]`), except for a single initial memory load per row. This can reduce latency and utilize the CPU pipeline better [Goumas et al. 2008].

GenOSKI

This method analyzes the matrix to find the patterns of nonzero entries in each block of size $r \times c$ and, for each pattern, generates straight-line code [Belgin et al. 2011]. A motivation of this method is to avoid the zero-fill problem of OSKI [Im et al. 2004] that generates efficient per-block code by inserting some zeros into the matrix data. GenOSKI generates one loop for each block pattern of nonzeros. A sample 4×4 block pattern and the corresponding code is given below, assuming there are 100 blocks with that pattern. The `rows` and `cols` arrays store indices of *blocks*, not individual nonzero elements. The index of a block is the location of the top-left corner of the block.

```
for (int a = 0, b = 0; a < 100; a++, b += 7) {
  int row = rows[a];
  int col = cols[a];
  w[row]   += vals[b]   * v[col+1] + vals[b+1] * v[col+3];
  w[row+1] += vals[b+2] * v[col]   + vals[b+3] * v[col+2] + vals[b+4] * v[col+3];
  w[row+2] += vals[b+5] * v[col+1];
  w[row+3] += vals[b+6] * v[col+3];
}
rows += 100;  cols += 100;  vals += 100*7; // Set the pointers for the next loop.
```
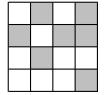
Data order: GenOSKI reorders matrix data to group blocks with the same pattern together. The accesses to w are sequential within a block but not across blocks.

Data size: Because this method stores indices of blocks, not individual nonzero elements, it can provide significant savings on the data size, unless there is a large number of very sparse blocks.

Code size: GenOSKI generates one for-loop for each block pattern, and the body of a loop contains as many multiplications as the length of the pattern. Hence, the code size is proportional to the number of block patterns and the sum of their lengths.

GenOSKI often performs well, especially when most blocks are fairly dense. This is because (1) locality within blocks is improved, (2) matrix data are usually reduced, and (3) there is room for compiler optimizations in for-loop bodies. Similarly to RowPattern, GenOSKI also eliminates indirect indexing on v. Nevertheless, this method may greatly increase the number of writes into the output vector w; the other methods write each w element only once. For the evaluation in this paper, we use blocks of size 4×4 and

5×5, as these were the block sizes that obtained the best performance in our previous study. We abbreviate these as **GenOSKI44** and **GenOSKI55**, respectively.

Unfolding

This method completely unfolds the CSR loop and produces a straight-line program that consists of a long sequence of assignment statements of the form

```
w[i] += A_{i,j_0} * v[j_0] + A_{i,j_1} * v[j_1] + ...;
```

where the italicized parts—i, A_{i,j_0}, j_0, and so on—are *fixed* values, not variables or subscripted arrays. This method eliminates the need to store `rows` or `cols` arrays separately because all the matrix information is implicit in the code. It also produces the lowest number of executed instructions but should produce, by far, the longest code. The size of the code is proportional to NZ. For this reason, it is not expected to yield good performance usually. However, it occasionally beats the other methods substantially. We have measured Unfolding as the best method for 13–21 matrices of 610. For these matrices, Unfolding's performance was on the average 1.23–1.35× of the performance of the *second*-best method. The ratio goes as high as 2.52×. These results show that Unfolding is not the winner method in most of the time, but when it is, its performance may substantially exceed the other methods. Therefore we decided to include Unfolding among the specialization methods we evaluate. It is also an interesting case from the point of view of machine learning to include a class that does not have many samples.

The main reason why Unfolding may yield very good performance is the repeated nonzero values of the matrix. To see why, suppose the following statements are produced after unfolding the SpMV loop, where 1.1 and 2.2 are matrix values:

```
w[0] += 1.1 * v[3] + 2.2 * v[4] + 1.1 * v[9];
w[1] += 2.2 * v[4] + 1.1 * v[9];
```

Compilers (we experimented with icc, clang, and gcc) tend to put only the *unique* floating point values into the data section and load values from there. Since the nonzero values of the matrix are available, this is a valid optimization. Furthermore, the nonzero values can be loaded into registers once and reused multiple times. Hence, the statements are compiled as if the code were

```
double M[] = {1.1, 2.2};
register double m0 = M[0];
register double m1 = M[1];
w[0] += m0 * v[3] + m1 * v[4] + m0 * v[9];
w[1] += m1 * v[4] + m0 * v[9];
```

In effect, using a pool of unique values may significantly reduce the memory traffic required to transfer nonzero values and open up more space in the cache for other data. This optimization was studied previously by Kourtis et al. [2010] as "Value Compression." We also reported the impact of unique values on the performance in a previous work [Kamin et al. 2014].

Unfolding also enables arithmetic optimizations because nonzero values become explicit in the code. An expression of the form e + 1.0 * v[i] can be simplified to e + v[i], and e + -1.0 * v[i] can be simplified to e − v[i]. Futhermore, the inverse of distribution of multiplication over addition can be performed. For example, 7.0 * v[6] + 7.0 * v[8] can be transformed into 7.0 * (v[6] + v[8]). These arithmetic optimizations decrease the total number of FP operations needed in SpMV. Having fewer unique values increases the opportunities for these optimizations.

Table I. The Impact of Optimizations Possible in Unfolding. Best Performing Method's Speedup is in **Bold** Font

Matrix		Unique values	MUL inst.	Memory traffic (MB) and Speedup wrt Baseline					
N	NZ			Baseline	CSRbyNZ	RowPattern	GenOSKI44	GenOSKI55	Unfolding
Andrews		29	60,000	9.0	9.0	14.7	9.8	9.8	9.1
60,000	410,077				1.32×	0.80×	1.00×	0.89×	**1.50×**
EAT_RS		91	42,333	6.6	8.3	11.5	7.9	8.0	6.4
23,219	325,592				1.00×	0.73×	0.78×	0.77×	**1.23×**
kron_g500-logn16		103	29,281	47.8	71.1	84.8	59.6	59.4	42.9
65,536	2,456,398				0.63×	0.49×	0.75×	0.76×	**1.03×**
Reuters911		165	14,856	3.0	4.3	5.3	3.6	3.7	2.9
13,332	148,038				0.97×	0.78×	0.82×	0.83×	**1.55×**
soc-sign-Slashdot081106		2	0	10.8	11.8	18.5	12.4	12.5	9.9
77,357	516,575				1.49×	0.68×	1.04×	1.04×	**1.87×**
delaunay_n21		6,291,408	6,291,407	155.8	154.8	240.6	130.3	138.1	237.9
2,097,152	6,291,408				0.81×	0.73×	0.40×	0.44×	**1.21×**
roadNet-CA		2,766,607	2,766,606	87.8	87.1	94.2	62.4	62.4	125.6
1,971,281	2,766,607				1.19×	0.55×	0.46×	0.52×	**1.50×**
af_5_k101		9,027,150	9,027,150	181.8	181.8	147.5	150.4	144.6	299.8
503,625	9,027,150				0.74×	1.07×	0.96×	**1.41×**	0.49×
torso3		3,121,632	4,429,042	89.4	89.4	80.5	82.2	80.3	147.2
259,156	4,429,042				1.08×	**1.17×**	0.98×	0.91×	0.50×

Finally, Unfolding also increases opportunities for Common Subexpression Elimination (CSE) when few distinct values exist. Consider the code snippet we used above. CSE can reduce the FP operations as follows:

```
double M[] = {1.1, 2.2};
register double m0 = M[0];
register double m1 = M[1];
double subExp = m1 * v[4] + m0 * v[9];
w[0] += m0 * v[3] + subExp;
w[1] += subExp;
```

In our code generator (detailed in Section 3), when using the Unfolding method, we create a pool of unique values if the matrix has sufficiently few distinct nonzero values. We set the threshold for this to 5000. We also do the arithmetic optimizations mentioned above. Our generator does not employ CSE.

To give concrete evidence of the impact of Unfolding optimizations, let us look at Table I. Here we give the number of rows (N), number of nonzero values (NZ), number of unique values, the number of MUL instructions generated by Unfolding, and "memory traffic" values for plain CSR format (Baseline) and the specialization methods. The memory traffic values imply the amount of data elements "touched" by the corresponding SpMV computation, according to the model in Gropp et al. [1999], which ignores the cache. So, in addition to the traffic incurred by the `rows`, `cols`, and `vals` arrays, whose elements are accessed once, we also include the data accesses to the input and output vectors `v` and `w`. This means, for each method, an additional traffic of NZ×8 is incurred because of the accesses to `v`. In Baseline, CSRbyNZ, RowPattern, and Unfolding, an element of the output vector `w` is accessed twice (once for read, once for write). This incurs an additional NE×8×2 bytes, where NE is the number of nonempty rows. For GenOSKI, the traffic incurred by the accesses to `w` is calculated according to the block patterns and the number of blocks. The traffic values for specialization methods also include the generated code size. In our previous work, we presented formulas to calculate the matrix data and code sizes for these methods [Kamin et al. 2014].

We show information for nine matrices in Table I. Unfolding gives the best performance for the first seven of these on turing (our testbed machine that has the Intel

CPU), using sequential execution. The best method for af_5_k101 is GenOSKI55; for torso3, it is RowPattern. The first five matrices have few unique elements while the other four have many. Normally, SpMV executes one multiplication instruction per each nonzero element. Hence, a naive unfolding would result in NZ-many MUL instructions in the code. However, due to the optimizations we explained before, the MUL instructions have been substantially reduced. An extreme case is soc-sign-Slashdot081106, where no MUL instruction remains in the generated code, because the matrix contains only 1 and −1 as its nonzero values. Also, due to creating a unique value pool, Unfolding's output is almost always smaller in terms of memory traffic when compared to the outputs of other methods. It is usually smaller than even the baseline. The reductions in the number of instructions and the size is only possible if the number of distinct values is small. The data for af_5_k101 and torso3 matrices illustrate this.

Finally, to our surprise, we have also observed that Unfolding gives the best performance for some matrices that have no or very few repeated values. The delaunay_n21 and roadNet-CA in Table I are two such matrices. Even though the optimizations we discussed above are not applicable to these matrices, Unfolding performs very well because it eliminates indirect indexes on the vector v and replaces them with constant indices (e.g., v[9]). A common property we observed in these matrices is that they are connectivity matrices that have a very large number of row patterns and a high number of sparse blocks. So RowPattern and GenOSKI do not perform well. Also, the average length of rows is very low (e.g., 3.0 in delaunay_n21, 1.4 in roadNet-CA). This causes loop overheads and branch prediction penalties in other methods.

We acknowledge that our list of methods is not complete. There exist many other matrix storage formats (e.g., ELL [Grimes et al. 1978] and DIA [Saad 2003]) that require no specialization yet may give better performance for some matrices. The problem is, covering all the possibilities seems practically impossible, as there is a very large number of formats and also hybrid combinations. So, we did not include generic storage formats except CSR in our evaluation; we specifically focused on specialization methods, and we limited ourselves to the SpMV methods presented here. That said, in our previous work [Kamin et al. 2014], we had compared our specialization methods with BiCSB [Buluç et al. 2011] and CSX [Kourtis et al. 2011]. The specialization methods we use in this paper had performed the best most of the time. We had also experimented with hybrid approaches but had not obtained high speedups.

3. CODE GENERATION APPROACH

We developed a special-purpose compiler that generates executable SpMV code at runtime. We do not generate source code, use scripts, or invoke an external compiler at runtime. The compiler takes a matrix and a method name as inputs and emits X86_64 object code into a memory buffer. The emitted code is dynamically loaded into the program and a function pointer is returned to the user.

For boilerplate tasks such as managing the object file format (e.g., arranging the code/data sections in the Elf, Mach-O formats) and dynamic loading, we use LLVM [Lattner and Adve 2004; LLVM 2013]. Instructions are emitted into LLVM's internal buffer at its machine-code layer. We do not generate any LLVM intermediate representation code but rather emit machine instructions directly—bit by bit—to avoid time-consuming compiler passes (e.g., alias analysis, register allocation, and global value numbering). We took this approach to minimize runtime code generation cost. The compiler is implemented in C++ to best integrate with the LLVM API.

Our compiler generates parallel code. For this, the matrix is split into as many partitions as the number of threads. Partitioning is row oriented and aims to assign roughly an equal number of nonzero values to each partition, using the following approach: If there are t threads, starting from the first row, then we assign consecutive

ALGORITHM 1: The Pseudo-Code of the CSRbyNZ Code Generator. This Generator Produces X86_64 Code for Each Distinct Row Length, Corresponding to the Source Snippet on Page 3.

```
// rows array is in %rdx, cols is in %rcx, vals is in %r8,
// v is in %rdi, w is in %rsi, a is in %rbx, b is in %r9
foreach row length L do
    M ← number of rows with row length L;
    emit(xor %rbx, %rbx);                                              // reset a to 0
    emit(xor %r9, %r9);                                                // reset b to 0
    emit(alignment to 16 bytes);                          // for better cache line utilization
    P ← current position in the object code buffer;
    emit(xor %xmm0, %xmm0);                                            // reset xmm0 to 0
    for i ← 0 to L do
        // Emit code to calculate vals[b+i]*v[cols[b+i]] and accumulate in %xmm0
        emit(mov i × 4(%rcx,%r9,4), %rax);                        // rax ← cols[b+i]
        emit(mov i × 8(%r8,%r9,8), %xmm1);                        // xmm1 ← vals[b+i]
        emit(mul (%rdi,%rax,8), %xmm1);                        // xmm1 ← xmm1 * v[rax]
        emit(add %xmm1, %xmm0);                                // xmm0 ← xmm0 + xmm1
    end
    emit(mov (%rdx,%rbx,4), %rax);                                // rax ← rows[a]
    emit(add L, %r9);                                                // b ← b + L
    emit(add 1, %rbx);                                                // a ← a + 1
    emit(add (%rsi,%rax,8), %xmm0);                        // xmm0 ← xmm0 + w[rax]
    emit(cmp M, %rbx);                                      // compare M and loop counter a
    emit(mov %xmm0, (%rsi,%rax,8));                            // w[rax] ← xmm0
    emit(jne P);                               // Jump to loop header if limit not reached
    emit(add M × 4, %rdx);                                      // rows ← rows + M
    emit(add M × L × 4, %rcx);                                // cols ← cols + M × L
    emit(add M × L × 8, %r8);                                // vals ← vals + M × L
end
```

rows to the first partition until the number of elements contained by the partition is at least nz/t. When the first partition has been given at least nz/t elements, we continue the same process for the next partition using the subsequent rows. This 1D partition is a common approach [Williams et al. 2009; Belgin et al. 2011; Byun et al. 2012; Liu et al. 2013]. For each partition, a function is generated using the specified specialization method. The generated functions are executed concurrently using OpenMP [2009]. Because partitioning is row oriented, no two threads share a common row. Hence, a locking mechanism or a final reduce-add operation is not needed.

When developing our purpose-built compiler, we naturally faced the problem of which machine instructions to use; that is, how to derive the assembly code. For this, we first generated code at source level and manually examined the assembly code produced by icc and clang (using the -O3 flag) to learn what instruction choices the compilers make. We focused on how the compilers compiled the loops similar to those we provided in Section 2. Although long, our code consists of replicating a straightforward loop structure over and over. We then wrote the code generator to match the output of compilers as closely as we could.

The way we generate assembly code is mostly straightforward. Algorithm 1 provides the CSRbyNZ code generator in pseudocode. This generator produces X86_64 machine code corresponding to the sample source code given for CSRbyNZ in Section 2. We wrote *emit* functions to write specific bits into the in-memory object code buffer for the given opcode and arguments. The X86_64 code generated by this CSRbyNZ generator for the t2em matrix is shown in Figure 1.

```
xorl  %r9d, %r9d
xorl  %ebx, %ebx
nopw  (%rax,%rax)
xorps %xmm0, %xmm0
movslq (%rcx,%r9,4), %rax
movsd (%r8,%r9,8), %xmm1
mulsd (%rdi,%rax,8), %xmm1
addsd %xmm1, %xmm0
movslq 4(%rcx,%r9,4), %rax
movsd 8(%r8,%r9,8), %xmm1
mulsd (%rdi,%rax,8), %xmm1
addsd %xmm1, %xmm0
movslq 8(%rcx,%r9,4), %rax
movsd 16(%r8,%r9,8), %xmm1
mulsd (%rdi,%rax,8), %xmm1
addsd %xmm1, %xmm0
```

```
movslq 12(%rcx,%r9,4), %rax
movsd 24(%r8,%r9,8), %xmm1
mulsd (%rdi,%rax,8), %xmm1
addsd %xmm1, %xmm0
movslq 16(%rcx,%r9,4), %rax
movsd 32(%r8,%r9,8), %xmm1
mulsd (%rdi,%rax,8), %xmm1
addsd %xmm1, %xmm0
movslq (%rdx,%rbx,4), %rax
addq $5, %r9
addq $1, %rbx
addsd (%rsi,%rax,8), %xmm0
cmpl $917300, %ebx
movsd %xmm0, (%rsi,%rax,8)
jne -140
addq $3669200, %rdx
addq $18346000, %rcx
addq $36692000, %r8
```

```
xorl  %r9d, %r9d
xorl  %ebx, %ebx
nopw  %cs
xorps %xmm0, %xmm0
movslq (%rcx,%r9,4), %rax
movsd (%r8,%r9,8), %xmm1
mulsd (%rdi,%rax,8), %xmm1
addsd %xmm1, %xmm0
movslq (%rdx,%rbx,4), %rax
addq $1, %r9
addq $1, %rbx
addsd (%rsi,%rax,8), %xmm0
cmpl $4332, %ebx
movsd %xmm0, (%rsi,%rax,8)
jne -52
addq $17328, %rdx
addq $17328, %rcx
addq $34656, %r8
```

Fig. 1. The CSRbyNZ code generated for t2em, a 921,632×921,632 matrix with 4,590,832 nonzeros. t2em has 917,300 rows whose length is 5 and 4,332 rows whose length is 1.
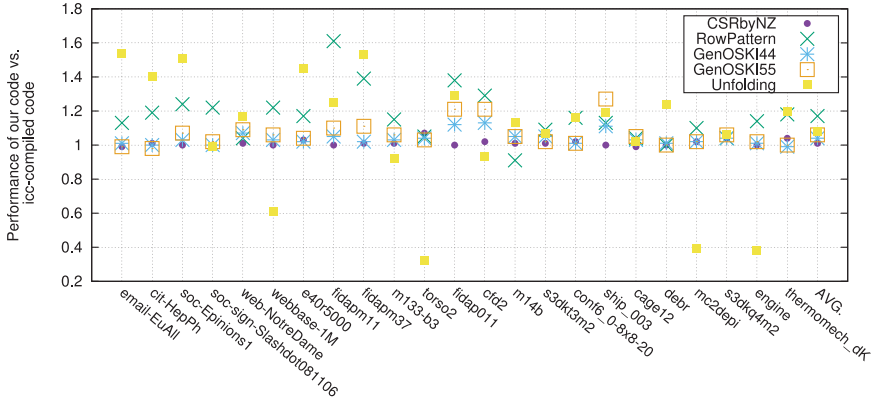

Fig. 2. The performance ratio of our compiler's output to icc's output for the matrices used in Kamin et al. [2014]. A value greater than 1 means we generated more efficient code than icc.

Our focus in this work is not generating the best SpMV code per se. We have not aggressively optimized the code we are generating; we are not doing optimizations such as vectorization, common subexpression elimination (CSE), or explicit prefetching.

Directly generating object code instead of going through the usual compiler passes makes the quality of our generated code questionable. To make sure that we generate efficient enough code, we compared our compiler's output with icc's. For this, we generated source code for all the 23 matrices that were used in Kamin et al. [2014]. We compiled these codes using icc with flags -O3 -no-vec (vectorization disabled, because our generator does not do vectorization). We measured the performance of the compiled code and compared against our code generator.

In Figure 2, we see the ratio of our code's performance to the performance of the code generated by icc. A value greater than 1 means our code performed better and smaller than 1 means icc's output performed better. The test was done on our Intel testbed machine using single-threaded execution. For CSRbyNZ, GenOSKI44, and GenOSKI55, the performances are consistently close, with our code performing slightly better: On average (last column in Figure 2), the ratios are 1.01, 1.04, and 1.06, respectively. For RowPattern, our code performs better than icc for 21 cases of 23. On the average, the

Table II. Performance of the Code Compiled by icc vs. Our Code Generator

Matrix	Best performing method when using		our code / icc
	our generator	icc	
soc-sign-Slashdot081106	Unfolding	Unfolding	0.99
webbase-1M	CSRByNZ	Unfolding	0.77
mc2depi	RowPattern	Unfolding	0.94
engine	Unfolding	Unfolding	0.38
fidapm37	GenOSKI55	GenOSKI44	1.07
18 other matrices	*Same for both*		1.04 (avg.)

ratio is 1.17, with a maximum of 1.61. Unlike other methods, Unfolding's performance varies with the input matrix greatly. The performance ratio for Unfolding ranges between 0.32 and 1.54 and is 1.08 on average.

Table II shows the best of the five specialization methods for the code generated by icc and our compiler. The last column gives the performance ratio between our compiler's winner and icc's winner. Again, a value larger than 1 means our code performs better. For 18 matrices of 23, the winning method for icc-compiled code and our compiler is the same. These codes perform similarly, with our compiler's output giving $1.04\times$ the performance of icc. For fidapm37, the winning method when using our generator is GenOSKI55, while it is GenOSKI44 with icc. The performances are close. There are four matrices that are worth more discussion: soc-sign-Slashdot081106, webbase-1M, mc2depi, and engine. In all of these, Unfolding is the winner among icc-compiled code. Our Unfolding performed very close to icc for soc-sign-Slashdot081106. This is a matrix that has only 1 and −1 as its nonzero values; we applied the arithmetic optimizations and so were able to match icc's performance. For engine, although Unfolding performs the best among the code generated by our compiler, it is significantly slower than icc-compiled Unfolding. Our compiler's Unfolding also could not meet the performance of icc's Unfolding for mc2depi and webbase-1M; other methods, RowPattern and CSRbyNZ, respectively, were the best. The performance of RowPattern for mc2depi was close to icc's Unfolding, but for webbase-1M there is a large gap. When we examined icc's Unfolding output for the matrices where icc outperforms our generator, we saw that icc applies optimizations that we do not do, such as common subexpression elimination (CSE) and instruction reordering.

In summary, the code that we generate, except for Unfolding, is either competitive with or better than icc's output. We were able to achieve this performance by generating code in a straightforward manner and without having to go through compiler phases, which are expensive at runtime. To give a measure, compiling the C source codes for 23 matrices took about 2 days on our testbed machine. Code generation has to be very rapid for runtime specialization to pay off. That is why we wrote our purpose-built compiler.

There are three dimensions of concern in runtime code generation in a setting like ours: (1) quality of the generated code, (2) speed of code generation, and (3) adaptability of the generator to new architectures. Achieving high levels in all three dimensions does not seem possible with the current state of the art. For instance, we could have followed a template-based approach (e.g., Consel et al. [2004]) to satisfy dimensions (2) and (3) but not dimension (1); compiling templates separately misses intertemplate optimization opportunities. We could have generated code at an AST or intermediate representation level (e.g., with Jumbo [Kamin et al. 2003] or LMS [Rompf and Odersky 2010]) and use an existing compiler back-end to optimize the generated program, but this would fail to satisfy dimension (2). We opted for dimensions (1) and (2) at the price of dimension (3): Our generator does not easily adapt to changes in the architecture.

To handle updates made to the target instruction set architecture, first, we would have to write new *emit* functions to support the new instructions. This is straightforward to do. Second, the code generator for each specialization method would have to be updated to use the new instruction emitting functions. This would have to be done by a programmer who knows where and under which conditions to use these new instructions. Because we do not outsource code generation to an external compiler, this step does not happen automatically and would be the most expensive part of the adaptation in terms of developer effort. This is a price we pay in exchange for quickly generating fast code.

4. AUTOTUNING RUNTIME SPECIALIZATION

Performance portability is a well-known challenge brought by the complexity of modern computer architecture. Autotuning has been successfully applied to solve this problem for HPC kernels including SpMV, dense linear algebra, and discrete Fourier transform [Frigo 1999; Whaley et al. 2001; Püschel et al. 2005; Vuduc et al. 2004, 2005; Muralidharan et al. 2014]. The same problem occurs in specialized SpMV code; the best-performing SpMV specialization method depends on both the matrix and the machine [Kamin et al. 2014]. In this section we discuss the use of autotuning to predict which method will perform the best for a given matrix. Prediction is important to avoid having to generate all the code variants and try them out, because runtime specialization has nontrivial cost (we discuss code generation costs in Section 7).

The autotuning process is as follows:

(1) At install time, code is generated for a set of training matrices using all the specialization methods. The generated programs as well as a nongenerative one (i.e., Intel MKL as the baseline) are executed and their performances are recorded.
(2) The collected data are used to train a multiclass classifier where several matrix properties are used as features (detailed below, in Section 4.2) and the names of the best performing methods are used as classes.
(3) At runtime, the user calls the library with a new matrix. Features are extracted from the matrix and are fed into the previously trained multiclass classifier. The classifier outputs a class, which denotes the method that is predicted to perform the best for the given matrix.
(4) SpMV code is generated using the predicted method if it involves specialization (the baseline method may have been predicted as well).
(5) A function pointer is returned to the user to be used for the subsequent SpMV operations for the given matrix.

In this work we evaluate how one can accurately predict the best SpMV method for a particular matrix. Our experimental results will show the prediction accuracy and the cost of runtime prediction and code generation; that is, when would specialization compensate its runtime overheads. We first discuss the impact of memory bandwidth and how this shapes the matrix features we chose for autotuning.

4.1. Memory Bandwidth

The performance of SpMV is highly affected by the amount of data transferred between CPU and memory [Gropp et al. 1999]. Nonspecialized methods usually have small codes; there the concern is the size of the matrix data. On one hand, specialization may reduce matrix data significantly. On the other hand, code may become very long. Both the matrix data size and the code size should be counted when talking about memory bandwidth, because code is also brought into the CPU from the memory. In Section 2 we commented on the code and data sizes implied by each method. In a previous work we used formulas to compute code and data size for the different methods [Kamin et al.

Table III. Number of Times a Method Yields the Smallest Size

Smallest (code + data)	CSRbyNZ	RowPattern	GenOSKI44	GenOSKI55	Unfolding
size occurrence	63	117	193	229	8

2014]. To measure the role of memory bandwidth, we calculated the code and data size for all the 610 matrices we use in this study. We then asked the question, "How would an autotuner perform if it always picked the method with the smallest data?" When compared to the speedup that could be achieved by a (hypothetical) perfect predictor that always picks the best performer, this smallest-size strategy yielded 86–91% of the achievable speedup. However, an SVM-based approach using the features we list in the next section obtains 97–99% of the achievable speedup (Section 6). In Table III, we show the number of times each method has the smallest size. CSRbyNZ is the smallest only 63 times, but it performs the best for many more matrices (see Figure 4 in Section 6). The opposite situation holds for GenOSKI methods. They yield the smallest size for many matrices but do not perform the best for that many cases.

This shows that even though memory is a dominant factor in SpMV performance, relying on only the size falls short of the achievable speedup. Table I also provides concrete examples of this argument. Another problem with the pick-the-smallest-size approach is that the total size of CSRbyNZ is most of the time slightly larger than the baseline. Hence, making a choice between CSRbyNZ and the baseline method solely based on size is insufficient. Other decision factors, such as the average length of rows or the number of distinct row lengths, are needed. At this point, one starts to feel the need of a model, and that is what the machine-learning based autotuning approach builds for us, based on the matrix features we provide and also the actual performance on machines. Hence, it also provides adaptation for a specific computer.

4.2. Features

We selected matrix features that indicate both the data and code size. We also picked features that hint at the number of iterations the generated loops execute. Table IV shows the feature set we are using. The features are classified based on the method that will have the highest impact from this feature. A total of 29 features are collected for each matrix (4 general structure, 4 CSRbyNZ, 8 RowPattern, 1 Unfolding, 6 GenOSKI44, and 6 GenOSKI55). We collect the number of rows (N), number of nonzeros (NZ), and nonzeros per row to represent the general structure of a matrix. We also include the number of nonempty rows (NE) because no code is generated for empty rows by RowPattern, CSRbyNZ, and Unfolding methods and some matrices have many empty rows. For instance, in our set of 610 matrices, 52 matrices have 10% or more empty rows; among these, 28 have more than 20% of their rows empty. From our point of view, Intel MKL is a black box, and we cannot have features specifically designed for it. This is yet another challenge for making successful predictions.

For CSRbyNZ, we collect the number of distinct row lengths, which indicates how many loops will be generated, and the sum of row lengths, which indicates how long the generated loop bodies will be. So, the first two features represent the code length for CSRbyNZ. The next two features are selected to indicate runtime. The average number of rows per each row length denotes how many times, on the average, each loop will iterate. The average of distinct row lengths indicates how long, on the average, a loop body will be; hence, it is an approximation of the runtime of one loop iteration.

There are corresponding features for RowPattern and GenOSKI. The number of patterns and the sum of pattern lengths indicate the code size. The average number of rows (respectively, blocks) per pattern, and the average length of patterns indicate the average runtimes of generated loops. RowPattern generates a loop for each

Table IV. Matrix Features Grouped under the Method They Impact the Most

General structure
Number of rows (N)
Number of nonzero elements (NZ)
Number of nonempty rows (NE)
Avg. number of nonzero elements per row (i.e. NZ / N)
CSRbyNZ
Number of distinct row lengths (RL)
Sum of distinct row lengths (SR)
Avg. number of rows for each row length (i.e. NE / RL)
Avg. of distinct row lengths (i.e. SR / RL)
RowPattern
Number of row patterns that apply to only a single row (R 1)
Number of row patterns that apply to multiple rows (R 2)
Sum of lengths of row patterns that apply to a single row (R 3)
Sum of lengths of row patterns that apply to multiple rows (R 4)
Avg. number of rows per row pattern that apply to multiple rows (R 5)
Avg. length of row patterns that apply to a single row (R 6)
Avg. length of row patterns that apply to multiple rows (R 7)
Ratio of NZ elements covered by *effective* row patterns (R 8)
Unfolding
Number of unique NZ values (capped at 5000) (U)
GenOSKI (for 4×4 and 5×5)
Number of block patterns (G 1)
Sum of lengths of block patterns (G 2)
Number of nonempty blocks (G 3)
Avg. number of blocks per block pattern (G 4)
Avg. length of block patterns (G 5)
Ratio of NZ elements covered by *effective* block patterns (G 6)

pattern; however, if a pattern is unique to only one row, completely unfolded code is generated. Therefore, we distinguish these cases when collecting RowPattern features. RowPattern and GenOSKI features also include the ratio of NZ elements covered by *effective* row patterns and block patterns, inspired from Belgin et al. [2011]. We say a row pattern is effective if its length is more than 3 and it covers at least 1000 NZ elements; a block pattern is effective if its length is more than 3 and it applies to at least 1000 blocks.

For GenOSKI, we collect the number of nonempty blocks. This denotes the total number of iterations generated loops will execute. The corresponding feature for CSRbyNZ and RowPattern is the number of nonempty rows, which is already in our list.

Unfolding's performance is highly sensitive to the number of distinct NZ values as discussed in Section 2. Hence, we have this value as a feature.

Before using for autotuning, we transformed the raw feature values as follows: (1) We took the *log* the values, because they show a skewed distribution. The effective block coverage (i.e., G 6) is the only exception to this. (2) We normalized the features to the [−1, 1] interval. This transformation is common in machine learning.

To the best of our knowledge, the features that we pick to indicate the code size are unique to our work. In existing work, features are usually determined according to the matrix storage formats, not code size. N and NZ are almost always collected as features (e.g., El Zein and Rendell [2012], Armstrong and Rendell [2008, 2010], Li et al. [2013], and Neelima et al. [2014]). NZ/N is also common [El Zein and Rendell 2012; Su and Keutzer 2012; Li et al. 2013]. Some other features used in the literature are

—zero-fill ratios for formats like DIA, ELL, and BELLPACK [Choi et al. 2010; Abu-Sufah and Abdel Karim 2013; Li et al. 2013],
—variation of row lengths [Abu-Sufah and Abdel Karim 2013; Armstrong and Rendell 2008, 2010; Li et al. 2013],
—mean neighbor count of nonzero elements [Armstrong and Rendell 2008, 2010],
—number of blocks and dense blocks per super row [Su and Keutzer 2012],
—number of diagonals, number of nonzero elements per diagonal [Su and Keutzer 2012; Li et al. 2013],
—max number of nonzeros per row [Li et al. 2013; Neelima et al. 2014], and
—memory traffic (number of bytes fetched, number of writes to `w`) [Belgin et al. 2011].

In an attempt to give more information to the learner, we experimented with other features. For instance, we decomposed the properties in the form of histograms to carry more fine-tuned information. For example, the number of row patterns whose length is less than 3, between 3 and 10, and more than 10, and so on (and similarly for CSRbyNZ and GenOSKI). We also used mean and standard deviation values. However, those attempts did not improve the prediction success and often decreased the quality, probably because of overfitting (also known as the curse of high dimensionality).

Full vs. Capped Feature Set

We call the features listed in Table IV the **full feature set**. In Section 6, we will see that the full feature set gives us good prediction success, but it is expensive to compute. As an alternative, we stop collecting some of the features when a certain cap is reached. We set this cap for RowPattern-related features at 2000 row patterns and for GenOSKI-related features at 5000 block patterns. We call this the **capped feature set**. The only difference between the full and the capped feature set is that when the cap value is reached, associated feature values are frozen and the matrix is no longer analyzed for those features. Analysis continues normally for other features. The number of distinct values is always capped at 5000.

The intuition behind the capped approach is that many matrices have too many row or block patterns. When this is the case, full analysis is expensive, because the set/map structures used for keeping track of the patterns become large. However, we observed that in general it is unlikely for RowPattern and GenOSKI to be the best method when there are too many patterns. So there is no need to do a complete analysis in this case. With the capped approach, many matrices will be only partially analyzed for RowPattern and GenOSKI. The features related to these methods will not always be the exact values. However, we saw that this inaccuracy causes only a slight decrease in the prediction success. In return, the feature extraction costs are reduced. We did not put a cap on CSRbyNZ features because the number of distinct row lengths is usually low and CSRbyNZ analysis is not expensive. Details are in Section 6.

We performed a correlation analysis between the features, shown in Figure 3. The correlations show that, in general, we have low redundancy among features. There is high correlation between N and NE (nonempty rows). This is because most of the matrices have elements on every row. However, there are some that have empty rows, and we want to distinguish them. (In our set of 610 matrices, 52 matrices have 10% or more and 28 have 20% or more of their rows empty.) So we kept NE in the features. We also see high correlation between the corresponding features of GenOSKI44 and GenOSKI55. This is not surprising since the two are instances of the same method. Finally, there is correlation between the number of patterns (respectively, distinct row lengths) and the sum of pattern lengths (respectively, sum of row lengths) in the RowPattern, GenOSKI, and CSRbyNZ methods. This is also normal; the sum of pattern lengths increases as the
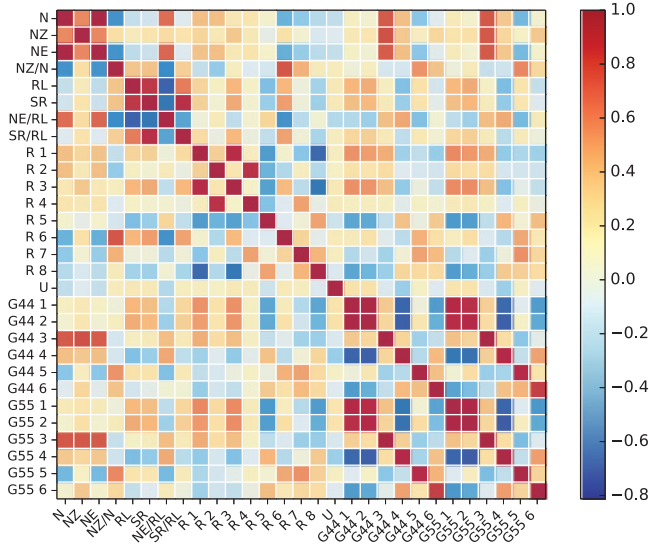
Fig. 3. Correlations between features of the full feature set.

number of patterns increases. We nevertheless kept these features in our set because they indicate important and separate properties about the generated code size.

We determined the set of features according to the specialization methods and the code generation approach. If a new method is added to the system, related features would have to be included. Similarly, changes in the architecture may trigger an update to the list. For instance, the ratio of consecutive column indices is potentially a useful matrix feature in case of vectorization.

4.3. Classifier

There are several multiclass classifiers to use as the learning model. We experimented with many, including Random Forest and Decision Tree Classifier. We found that C-Support Vector Classification (SVC) with an RBF (Gaussian) kernel gives the best results. We tried a variety of C and gamma parameters for RBF; we used the results for the parameter values that yielded the best prediction rates.

4.4. Classes

In the learning phase, the classifier is fed with the features and the *classes* of the matrices. The classifier uses these data to create a model that associates matrix features with the corresponding classes. We tried different approaches to specify the class:

Naive Approach: We used the best performing method for a matrix as its class. In this approach, there are as many classes as SpMV methods (six in our case). This naive definition of classes has a potential problem, though: It ignores the fact that methods may perform very close to each other. For example, suppose CSRbyNZ is the best method for a matrix, but RowPattern is also very good—good enough that, from the point of view of the user, picking RowPattern as the SpMV method instead of CSRbyNZ would also be acceptable. However, from the point of view of the classifier, picking RowPattern instead of CSRbyNZ is simply incorrect, because that is not the class that the matrix belongs to. In other words, defining the class of a matrix as its best method loses information about what other methods are also good choices. We observed that the average performance ratio of the best and the second-best methods

is 1.13–1.16$\times$ in our test setup. The ratio is less than 1.01$\times$ in 6–8% of the matrices, less than 1.02$\times$ in 12–16%, and less than 1.05$\times$ in 24–36%. We try to remedy this potential problem with the next approach.

Pair of Methods: We used the top two performing methods for a matrix as its class. So a class label is a *pair* of method names. In this approach, the prediction output of the classifier also contains *two* methods: a method predicted to be the winner and another that is predicted to be the runner-up. To decide which method to use for code generation, we ignore the runner-up and take the first method. To illustrate, let us take the previous example. There the matrix's actual class would be CSRbyNZ-RowPattern instead of just CSRbyNZ. If the classifier makes the prediction, say, CSRbyNZ-Baseline, then we generate code for CSRbyNZ. This is the best case for prediction. If the prediction is RowPattern-Unfolding or RowPattern-CSRbyNZ, then we generate code using the RowPattern method, which is not the best one but still a good choice.

Using the paired approach, more information is fed into the learner; however, a potential problem is that the number of possible classes increases significantly as compared to the naive approach. If M SpMV methods exist, then there are a maximum of $M \times (M - 1)$ classes. Having more classes may negatively impact the prediction's success because there will be fewer samples per class during the training phase, and there are more classes to distinguish from each other.

Another potential problem with the paired approach is that if the best method is substantially better than the second one, this information is not disseminated to the learner. To address this issue, we tried a variation of the paired labeling approach where we set a *threshold* value: If the best method is better than the second-best method by more than the threshold, then we repeated the best method also as the second method in the class name. For instance, suppose for some matrix CSRbyNZ is the best method, Unfolding is the second best, CSRbyNZ performs 1.30$\times$ of Unfolding, and the threshold value is 1.05$\times$. We labeled the matrix to be in the CSRbyNZ-CSRbyNZ class. This way we emphasized to the learner that for this matrix CSRbyNZ is really the best method. This approach introduces as many new classes as the number of methods.

Labeling happens automatically, with no human effort. For each matrix, the auto-tuner looks at the performance measurements of the SpMV methods and determines the class using the chosen approach. The results of the naive and paired labeling approaches are presented in Section 6. Thresholding did not sufficiently improve the prediction results; we give a brief discussion about this in Section 6.

5. EXPERIMENTAL SETUP

In our experimental evaluation, we use a set of 610 matrices obtained from the Matrix Market [1997] and the University of Florida collection [Davis and Hu 2011]. All our matrices are square and sparse. Their number of nonzero elements range from 100K to 15M, and dimensions range from 2K to 2.4M. Of the matrices, 129 are *pattern* matrices. In this case, the matrix data downloaded from the collection do not provide any nonzero values; only the positions of elements are stated. We populate such matrices with distinct values. Some matrices are symmetric, but we ignore this property.

Several of the matrices in our set are compiled from previously published papers [Buluç et al. 2009; Kourtis et al. 2011; Williams et al. 2009]. Others are arbitrarily chosen from the matrix collections without any specific criteria except that we preferred the matrices not to have more than 15M nonzeros to make the experiments runnable in a reasonable amount of time. The matrices come from a variety of domains, including circuit simulation, duplicate model reduction, electromagnetics, quantum chemistry, power network, computer graphics, and so on.

Table V. Target Platforms

| Name | Processor @ Freq (GHz) (Microarchitecture) | Cores | Cache Sizes (Bytes) | | | Mem (GB) | Linux OS | compiler |
			L1 (I/D)	L2	L3			
turing	Intel® Xeon® E5-2620 @ 2.00 (SandyBridge)	6	32K	256K	15M	16	Ubuntu 12.04	icc 14.0
milner	AMD FX-8350 @ 4.00 (Piledriver)	8	64K/16K	2M	8M	8	ArchLinux 3.14.4	gcc 4.8.2

We executed code on two unloaded X86_64 machines, one with an Intel and the other with an AMD processor. The properties of our testbed computers are in Table V. On both machines we generated code using five specialization methods (CSRbyNZ, RowPattern, GenOSKI44, GenOSKI55, Unfolding). We also collected the runtime of Intel MKL's SpMV function, and we use Intel MKL as the baseline when we calculate speedups. So, in total, six SpMV methods are used on the machines. We have also run the benchmarks on a third computer with an Intel Xeon E3-1220 CPU and found the results to be similar to turing; we do not include that machine's timings here.

We collected the running times as follows: For each matrix and SpMV code, we measured the time it takes to run the code for a few hundreds or thousands of times. The number of iterations is determined according to the matrix size, but we made sure that the measured time is long enough (e.g., at least 2s) to avoid fluctuation. We then divided the measured time by the number of iterations to find the running time of one SpMV operation. We repeated this test 3 times and took the lowest time (i.e., the fastest execution time) with the intuition that it reflects the execution with the least interference from external events. We measured feature collection, matrix conversion, and code generation times again by running them 3 times and taking the smallest measurement. We executed SpMV code both sequentially and in parallel. For parallel executions, we set the number of threads to be equal to the number of CPU cores (six on turing and eight on milner). We refer to the sequential runs as **turing-1** and **milner-1** and parallel runs as **turing-6** and **milner-8**.

For prediction experiments, we used the scikit-learn module of Python (version 2.7.9) [Pedregosa et al. 2011]. We applied 10-fold cross validation for training and testing. This is a standard approach in machine learning. We first shuffled the data and then split into 10 groups, each comprising of 61 matrices. For each group, training is done using the other 9 groups (549 matrices). The chosen group is used for testing whether the predictions made by the trained classifier is correct.

We used Principal Component Analysis (PCA), a technique in machine learning, to reduce the number of features in order to assist the classifier by supplying more correlated data, but we did not observe any improvement in the quality of predictions. Thus, the results we report do not include any application of PCA.

6. EXPERIMENTAL RESULTS

In this section we discuss the prediction results of the classifier. Figure 4 shows the distribution of best methods. Figure 5 shows the distribution of class labels when using the paired approach. In Figure 6 we show the prediction results for turing-1, turing-6, milner-1, and milner-8. For each, we show the number of *correct*, *semicorrect*, *incorrect*, and *bad* predictions (definitions given below), as well as the average speedup achieved when using the predicted methods (on top of each bar). We tried all four combinations of naive/paired labeling and full/capped feature sets.

In the *naive* class labeling approach, a single method name is used as the class of a matrix. Hence, if the autotuner's classification for a given matrix is the same as
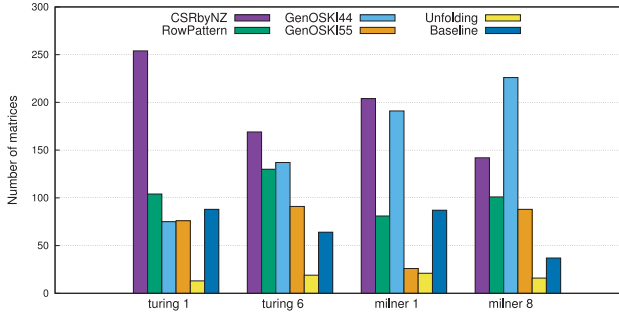
Test Name	Avg. Speedup with the best method
turing-1	1.33
turing-6	1.45
milner-1	1.39
milner-8	1.83

Fig. 4.   Left: Number of times each method is the best (610 matrices in total). Right: Average speedup w.r.t. the baseline performance when using the best method for each matrix.
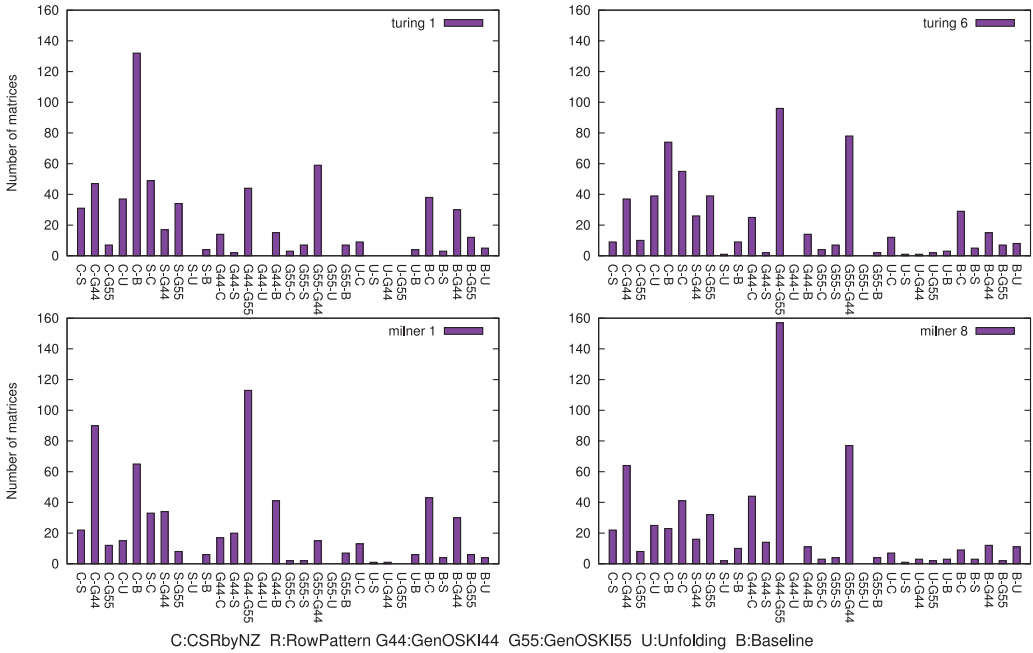

Fig. 5.   Class labels and corresponding counts for 610 matrices using the paired approach.

the actual best method, then it is a *correct* prediction. Otherwise it is an *incorrect* prediction.

In the *paired* class labeling approach, two method names are used as the class of a matrix. The autotuner's classification output is hence a pair of method names. As previously explained, we take the first method as the predicted one and ignore the second. If this first method is the same as the actual best method, then we categorize this prediction as *correct*; if it is the same as the actual second best method, then we categorize this prediction as *semicorrect*. Otherwise, the prediction is considered *incorrect*. In both naive and paired labeling approach, an incorrect or semicorrect prediction may have worse performance than the baseline. We call this a *bad prediction*.

We achieve average speedups of 1.31, 1.41, 1.37, and 1.77 when using paired labeling and the capped feature set (P-C bars in Figure 6). The speedups are slightly better when using the naive approach or the full set. Recall from Figure 4 that if always
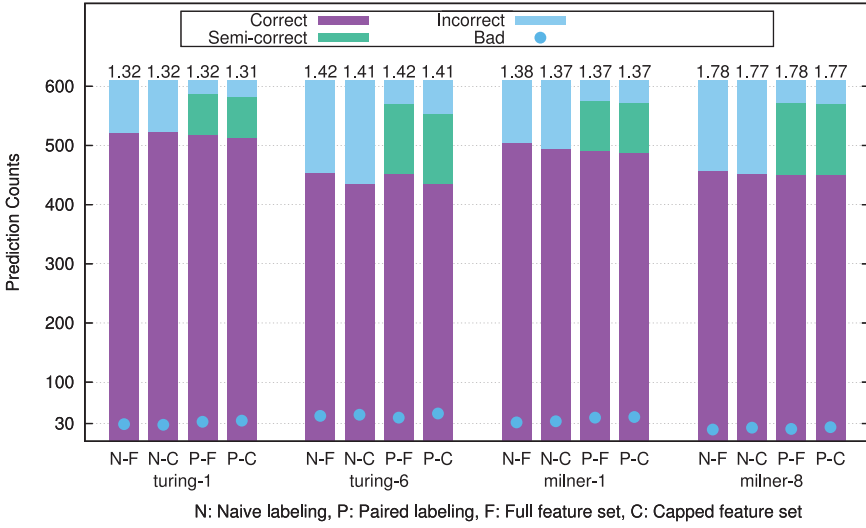
Fig. 6. Prediction results.

the best methods are used, the speedups are 1.33, 1.45, 1.39, and 1.83, respectively. So predictions obtain 97–99% of the maximum speedups. The best method can be predicted in 71–86% of the matrices, and the second-best method can be predicted in 11–20% of the matrices. Only 5–8% of the predictions choose a method worse than the baseline.

Full vs. Capped Feature Set
The full feature set gives only slightly better predictions than the capped feature set. The average speedups are either the same or only differ by 0.01. Taking into account that the capped feature set can be extracted faster than the full set (detailed in the next section), we favor the capped set and consider the marginal loss in the quality of predictions an acceptable tradeoff.

Naive vs. Paired Labeling
The naive and paired approaches yield similar speedups and prediction accuracy. The advantage of the paired approach to the naive approach is the confidence it provides from the machine learning (ML) point of view. Even though good speedup is achieved with naive labeling, about 14–29% of the predictions are "incorrect." This would make a machine-learning-savvy person feel uncomfortable; a success rate of about 70% is not considered the best in the ML community. By using the paired approach, we relax the definition of class labels and feed more information into the learner. This gives more confidence that the achieved speedups are good not just by luck.

Thresholding
We also experimented with the thresholding approach presented in Section 4. We used 1.01, 1.02, 1.03, 1.05, 1.10, and 1.15 as the threshold values. Usually, using the threshold yielded slight improvement in terms of correct predictions (approximately five more) and bad predictions (approximately four fewer). The achieved speedups did not change. However, the number of semicorrect and incorrect predictions were altered significantly. For instance, for turing-1, we obtained 69 semicorrect and 23 incorrect predictions when a threshold is *not* used but 19 semicorrect and 67 incorrect when a threshold value of 1.02 is used. This is because some classes contain repeated method names (e.g., CSRbyNZ-CSRbyNZ) when a threshold is used. For those classes, there is no chance for a semicorrect prediction; a prediction is either correct or incorrect,

according to our definition. For this reason, we decided not to use the thresholding approach.

7. LATENCY

SpMV specialization is likely to occur at runtime, unless the matrix (or at least its pattern) is available offline. If the matrix data are available only at runtime, the SpMV library has to be quick in producing the specialized function in order for specialization to bring profit. In this section, we discuss the issue of *latency*: How much time needs to be spent for prediction and code generation? How many SpMV iterations should be taken so specialization compensates its costs and starts to bring benefits? We show that, on average, the total cost of specialization is equivalent to 58 and 53 calls to the baseline SpMV operations, respectively, on two machines where we ran our experiments. For the matrices for which the predicted method brings $1.1\times$ or better speedup, we obtained average break-even points of 272 and 237 baseline SpMV operations on our testbed computers. These costs and break-even points are low enough that runtime specialization of SpMV for many real-world matrices in practical applications of iterative solvers is feasible.

In our SpMV library, we assume we are given a matrix defined in the standard Compressed Sparse Row (CSR) format. SpMV specialization for a matrix and a particular specialization method involves the following steps:

—*Matrix analysis:* Before generating code, the matrix is analyzed to collect method-related information, for example, what block patterns exist and which blocks have which patterns in GenOSKI. The result of matrix analysis is used for matrix conversion (next step), and when emitting instructions (the step after), for example, for each block pattern in GenOSKI, a loop is generated.
—*Matrix conversion:* The matrix data are converted from CSR format to the format needed by the particular specialization method. This usually involves reordering the matrix data.
—*Instruction emission:* X86_64 instructions are emitted in accordance with the specialization method, using the code generation approach explained in Section 3.
—*Boiler-plate:* A number of low-level tasks need to be carried out to execute the emitted code at runtime. These tasks include creating a target-specific (e.g., Mach-O or Elf) in-memory buffer to emit the instructions and dynamically loading this buffer for runtime execution. For these tasks, we use LLVM's machine-code layer.

Average costs of the code generation steps in terms of one baseline SpMV operation are given in Table VI. We provide two costs, "if best" and "overall," for each method. The "Overall" column gives the cost averaged over the whole set of matrices; "if best" gives the cost averaged over the matrices for which the particular method is the best performer. We see that, in general, costs are lower when the method happens to be the best. This is because shorter codes are often better than long codes, and the short code is generated quicker. For instance, if there is a large number of row patterns in a matrix, then both the analysis, instruction emission, and boiler-plate steps take a significantly longer time. A similar observation can be made for GenOSKI and Unfolding as well. Compared to the other methods, CSRbyNZ is usually very fast to analyze and generate.

Table VI also provides the costs for extraction of full and capped feature sets, as well as *end-to-end specialization*. The full feature extraction cost of a matrix is *less* than the sum of CSRbyNZ, RowPattern, Unfolding, GenOSKI44, and GenOSKI55 matrix analysis costs, because feature extraction tracks less data than matrix analysis. For instance, while the feature extraction step collects only the counts of patterns and blocks for GenOSKI, matrix analysis also needs to collect which patterns apply to which blocks. *End-to-end specialization* is calculated as

Table VI. Costs of Code Generation Steps and Feature Extraction
in Terms of One Baseline SpMV Operation

	turing-1		milner-1	
	if best	*overall*	*if best*	*overall*
CSRbyNZ				
Analysis	1.7	1.3	1.0	0.8
Conversion	3.6	4.6	3.4	4.2
Emission	0.8	2.0	0.7	1.5
Boiler-plate	1.1	2.3	1.0	1.8
RowPattern				
Analysis	19.4	27.9	16.2	20.5
Conversion	3.9	5.6	3.0	4.4
Emission	1.5	31.4	1.0	21.4
Boiler-plate	2.1	25.1	1.5	18.8
GenOSKI44				
Analysis	34.3	40.9	29.0	30.5
Conversion	3.2	3.2	3.4	3.2
Emission	1.5	2.3	0.9	1.7
Boiler-plate	1.3	1.6	1.0	1.1
GenOSKI55				
Analysis	38.0	42.7	27.3	32.2
Conversion	3.5	3.3	3.0	3.2
Emission	2.0	6.6	0.8	4.9
Boiler-plate	1.6	3.0	0.9	2.1
Unfolding				
Analysis	3.0	4.0	2.3	3.3
Conversion	0.0	0.0	0.0	0.0
Emission	60.6	108.6	44.0	77.0
Boiler-plate	13.1	38.4	10.6	28.5
Full feature set				
Extraction	71.2		57.0	
End-to-end specialization	90.3		75.8	
Capped feature set				
Extraction	39.0		34.6	
End-to-end specialization	58.0		52.9	

Feature extraction + Predicted method's (Analysis + Conversion + Emission + Boiler-plate).

When calculating end-to-end specialization, we take the analysis cost as zero if the predicted method is CSRbyNZ or Unfolding, because the needed information is already computed during feature extraction. The feature extraction and end-to-end specialization costs we report are averaged over all matrices.

The cost of end-to-end specialization is equivalent to 58.0 baseline SpMV calls on turing and 52.9 on milner when using the capped feature set. This means, even when the baseline method or a method whose performance is very close to the baseline is predicted, the amount of work that is spent due to specialization is about 50–60 iterations of SpMV. Considering that several hundreds of iterations in iterative solvers is typical, this may be an acceptable tradeoff.

Table VI shows values for only single-threaded execution, because we did not parallelize code generation phases yet. The boiler-plate step is delegated to LLVM, and we
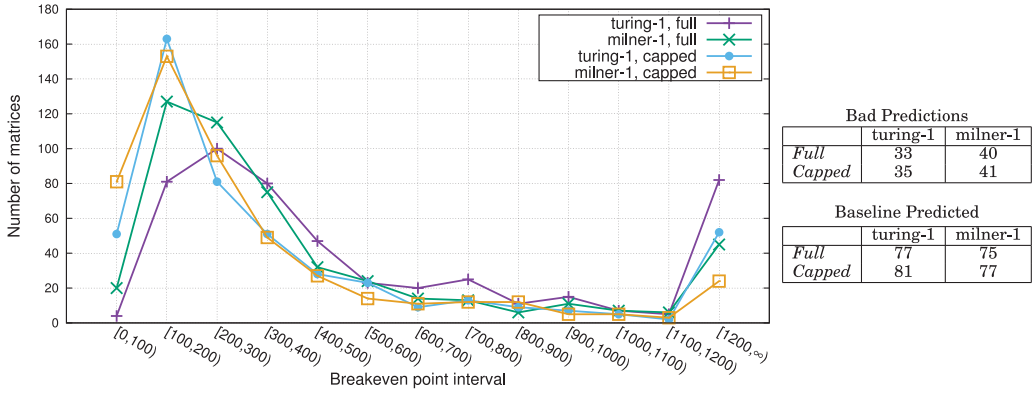
Fig. 7.   Distribution of break-even points of the predicted methods.

Table VII. Count and Break-Even Points of Predictions that Yield 1.1x or Better Speedup

	Full feature set		Capped feature set	
	turing-1	milner-1	turing-1	milner-1
No. of predictions with $\geq 1.1\times$ speedup	417	435	414	432
Avg. break-even point of predictions with $\geq 1.1\times$ speedup	406	314	272	237

are not sure if it can be parallelized, but it is possible for all the other steps to run concurrently by splitting the matrix and the analysis data into partitions.

Considering the end-to-end specialization costs, we calculate the *break-even point* for each matrix: How many times should we have to iterate SpMV so specialization compensates its cost and starts to bring advantage over the baseline implementation? Figure 7 shows the distribution of break-even points. The values in this figure have been prepared according to the predictions made using the paired labeling approach. The number of iterations used in iterative solvers depends on the desired accuracy of the solution, but several hundred or a few thousand is common in practice. Considering this fact, the break-even points shown in Figure 7, in particular those when the capped feature set is used, are practically useful, as for many matrices speedup would be gained. Note that for some matrices, the baseline method is predicted. For those matrices, no break-even point exists and no cost other than the feature extraction has to be paid. The bad predictions are the cases for when the predicted method performs worse than the baseline. For these cases, the library may simply default back to using the baseline implementation after detecting that the generated code performs poorly.

Belgin et al. report average break-even points from 500 to 700 *excluding* code generation cost in their work where they introduce the pattern-based representation (PBR) for SpMV [2011]. They report these break-even points for matrices for which at least $1.1\times$ speedup was observed (39 of 53 matrices). Because we use different methods and our matrix set is not the same (we have 610), our numbers are not directly comparable to theirs. However, to give a similar evaluation, our average break-even point for predictions that yield at least $1.1\times$ speedup (when the capped feature set is used and code generation cost is *included*) is 272 on turing (414 cases of 610) and 237 on milner (432 cases of 610), also shown in Table VII. Belgin et al. generate code by writing C files on the disk and invoking a compiler. Therefore, when runtime code generation is included, their break-even points increase to several thousands. Our code emission costs are much smaller, due to our purpose-built code generator.

8. RELATED WORK

Previous autotuning approaches for SpMV focus on choosing an optimal storage format, because even the basic sparsity regime of a matrix can have a profound effect on the performance [Bell and Garland 2009]. To this end, there exist work using decision trees [Li et al. 2013], dynamic-programming [Guo et al. 2014], reinforcement learning [Armstrong and Rendell 2008], heuristic-based autotuning [Abu-Sufah and Abdel Karim 2013], and model-driven approaches [Neelima et al. 2014; Choi et al. 2010]. To the best of our knowledge, ours is the first study on applying autotuning to pick among several specialization methods. This is challenging as the generated code structure also needs to be considered in addition to the data format. We used a SVM-based approach for autotuning. SVM is used in many autotuning systems including the Nitro framework [Muralidharan et al. 2014]. Recently, a two-level approach to autotuning was shown to be effective in addressing the complexities of mapping features to algorithmic configurations [Ding et al. 2015]. We leave it for future work to see whether this approach improves the prediction accuracy for our experiments. Most of the other autotuning work have smaller matrix sets than ours, for example, ~14–150 [Grewe and Lokhmotov 2011; Muralidharan et al. 2014; Neelima et al. 2014; Guo et al. 2014]. There also are studies with bigger matrix sets, for example, ~2000 in Li et al. [2013] and 1000 (synthetic) in Armstrong and Rendell [2008].

There exist several works that employ runtime specialization for SpMV. Willcock and Lumsdaine [2006] generate matrix-specific compression/decompression functions. Kourtis et al. [2011] also study data compression; they generate specialized SpMV routines for their CSX format in the LLVM intermediate representation. We, too, use LLVM, but only for boiler-plate tasks regarding object file management. They employ matrix sampling to reduce analysis costs by allowing minor loss in speedups; we use capped analysis for the same purpose. Sun et al. [2011] introduce a runtime code generator for OpenCL that produces code variants for diagonal patterns for their CRSD format. Belgin et al. [2011] propose a new format PBR which identifies recurring block structures that share the same pattern of nonzeros within a matrix. (The GenOSKI method we use is a variant of PBR.) A runtime code generator produces optimized custom kernel for each pattern. They generate source-level code and invoke an external compiler. They also have a code cache that can be used to dynamically link object files for already-compiled code. They show that priming this cache with common block pattern code reduces runtime costs. Mateev et al. [2000] introduce a generic programming API to generate efficient sparse code using high-level algorithms and sparse matrix format specifications. A similar work is presented in Grewe and Lokhmotov [2011], where efficient and system-specific SpMV kernels for GPUs are generated based on a storage format description. While this line of research generates code according to storage formats, we specialize code for a specific matrix.

Code generation for SpMV or related problems (i.e., matrix multiplication and vector dot product) is found as a case study in several previous papers. Fabius [Lee and Leone 1996] is a compiler that generates native code at runtime by deriving the generator from source code that contains binding-time annotations. Carette and Kiselyov [2011] show how to eliminate abstraction overheads from generic programs using multistage programming on Gaussian elimination. Rompf et al. [2013] combine various compiler extension techniques to generate high-performance low-level code. They demonstrate optimization of operations on sparse matrices, loop unrolling, and loop parallelization. SpMV, in the context of Hidden Markov Models, was also proposed as a Shonan Challenge [Aktemur et al. 2013].

We developed our code generator manually. It would be possible to derive it systematically from source code using a code generation/staging approach as in Fabius

[Lee and Leone 1996], LMS [Rompf and Odersky 2010], or Tempo [Consel et al. 2004]. However, we would either compromise the efficiency of the generated code or the speed of generation, as discussed in Section 3.

Earlier examples of using code generation to optimize linear algebra operations include Gustavson et al. [1970] and Fukui et al. [1989]. They generate machine code based on the matrix structure. Giorgi and Vialla [2014] generate SpMV kernels based on characteristics of the input matrix. Venkat et al. [2015] address indirect loop indexing and irregular data accesses in SpMV kernels and introduce new compiler transformations and automatically generated runtime inspectors. Our RowPattern method also eliminates indirect indexing. Neither of these papers do runtime generation.

9. CONCLUSIONS

In this paper we have shown that it is possible to use runtime specialization to form efficient SpMV when the same matrix is multiplied by many vectors. We have developed an end-to-end special-purpose compiler that generates efficient SpMV code which is specialized for a given matrix. Our compiler directly emits machine instructions without going through any intermediate representation to avoid time-consuming compiler passes. We took this approach to minimize runtime code generation cost.

We experimented with five specialization methods and also Intel MKL. We used multiclass classification and two class labeling approaches to predict a best method. Our experiments on two different machines using 610 matrices show that for 91–96% of the matrices, either the best or the second best method can be predicted. For autotuning, we used 29 matrix features; several of these are unique to our work. We used a capped feature extraction approach to reduce matrix preprocessing costs. We show that end-to-end specialization costs are equivalent to 53–58 baseline SpMV operations on the average. These costs are low enough that runtime specialization of SpMV for many real-world matrices in practical applications of iterative solvers is feasible.

REFERENCES

W. Abu-Sufah and A. Abdel Karim. 2013. Auto-tuning of sparse matrix-vector multiplication on graphics processors. In *Supercomputing*. Lecture Notes in Computer Science, Vol. 7905. Springer, Berlin, 151–164.

ACML. 2013. AMD Core Math Library User Guide 6.0.6. Retrieved from http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/acml.pdf.

B. Aktemur, Y. Kameyama, O. Kiselyov, and C. Shan. 2013. Shonan challenge for generative programming. In *Partial Evaluation and Program Manipulation (PEPM'13)*. 147–154.

W. Armstrong and A. P. Rendell. 2008. Reinforcement learning for automated performance tuning. In *Cluster Computing*. 411–420.

W. Armstrong and A. Rendell. 2010. Runtime sparse matrix format selection. *Procedia Comput. Sci.* 1, 1 (2010), 135–144.

M. Belgin, G. Back, and C. J. Ribbens. 2011. A library for pattern-based sparse matrix vector multiply. *Int. J. Parallel Program.* 39, 1 (2011), 62–87.

N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *High Performance Computing Networking, Storage and Analysis (SC'09)*. 18:1–18:11.

A. Buluç, J. Fineman, M. Frigo, J. Gilbert, and C. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *21st Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'09)*. 233–244.

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

J. Byun, R. Lin, K. Yelick, and J. Demmel. 2012. *Autotuning Sparse Matrix-Vector Multiplication for Multicore*. Technical Report UCB/EECS-2012-215. EECS Department, U. of California, Berkeley.

J. Carette and O. Kiselyov. 2011. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76, 5 (May 2011), 349–375.

J. Choi, A. Singh, and R. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Principles and Practice of Parallel Programming (PPoPP'10)*. 115–126.

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

T. Davis and Y. Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.

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

Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *Prog. Language Design and Implementation (PLDI'15)*. 379–390.

A. El Zein and A. Rendell. 2012. Generating optimal CUDA sparse matrixvector product implementations for evolving GPU hardware. *Concurr. Comput. Pract. Exper.* 24, 1 (2012), 3–13.

M. Frigo. 1999. A fast fourier transform compiler. In *Programming Language Design and Implementation (PLDI'99)*. 169–180.

Y. Fukui, H. Yoshida, and S. Higono. 1989. Supercomputing of circuits simulation. In *Supercomputing (SC'89)*. 81–85.

P. Giorgi and B. Vialla. 2014. Generating optimized sparse matrix vector product over finite fields. In *Mathematical Software (ICMS'14)*. 685–690.

G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. 2008. Understanding the performance of sparse matrix-vector multiplication. In *Parallel, Distributed and Network-Based Processing (PDP'08)*. 283–292.

D. Grewe and A. Lokhmotov. 2011. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *General Purpose Processing on Graphics Processing Units (GPGPU-4)*. 12:1–12:8.

Roger G. Grimes, David R. Kincaid, and David M. Young. 1978. *ITPACK 2.0 User's Guide*. Report CNA-150. Center for Numerical Analysis, University of Texas at Austin, Austin, TX.

W. Gropp, D. Kaushik, D. Keyes, and B. Smith. 1999. Toward realistic performance bounds for implicit CFD codes. In *Parallel CFD'99*.

P. Guo, L. Wang, and P. Chen. 2014. A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs. *IEEE TPDS* 25, 5 (May 2014), 1112–1123.

F. Gustavson, W. Liniger, and R. Willoughby. 1970. Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. *J. ACM* 17, 1 (Jan. 1970), 87–109.

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

A. Jain. 2008. *pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures*. Master's thesis. University of California, Berkeley.

S. Kamin, L. Clausen, and A. Jarvis. 2003. Jumbo: Run-time code generation for java and its applications. In *Code Generation and Optimization (CGO'03)*. 48–56.

S. Kamin, M. Garzarán, B. Aktemur, D. Xu, B. Yılmaz, and Z. Chen. 2014. Optimization by runtime specialization for sparse matrix-vector multiplication. In *Generative Programming: Concepts and Experiences (GPCE'14)*. 93–102.

K. Kourtis, G. Goumas, and N. Koziris. 2010. Exploiting compression opportunities to improve SpMxV performance on shared memory systems. *ACM Trans. Archit. Code Optim.* 7, 3 (Dec. 2010), 16:1–16:31.

K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris. 2011. CSX: An extended compression format for SpMV on shared memory systems. *SIGPLAN Not.* 46, 8 (Feb. 2011), 247–256.

C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO'04)*. 75–86.

P. Lee and M. Leone. 1996. Optimizing ML with run-time code generation. In *Programming Language Design and Implementation (PLDI'96)*. 137–148.

J. Li, G. Tan, M. Chen, and N. Sun. 2013. SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication. *SIGPLAN Not.* 48, 6 (June 2013), 117–126.

X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Supercomputing (ICS'13)*. 273–282.

LLVM. 2013. LLVM Web Site. Retrieved from http://llvm.cs.uiuc.edu.

N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. 2000. Next-generation generic programming and its application to sparse matrix computations. In *Supercomputing (ICS'00)*. 88–99.

Matrix Market. 1997. Matrix Market Web Site. Retrieved from http://math.nist.gov/MatrixMarket.

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

MKL. 2013. Intel® Math Kernel Library. Retrieved from http://software.intel.com/en-us/articles/intel-mkl.

S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. 2014. Nitro: A framework for adaptive code variant tuning. In *Parallel and Distributed Processing Symp. (IPDPS'14)*. 501–512.

B. Neelima, G. Ram, M. Reddy, and Prakash S. Raghavendra. 2014. Predicting an optimal sparse matrix format for SpMV computation on GPU. In *Parallel & Distributed Processing Symp. Workshops (IPDPSW'14)*. 1427–1436.

OpenMP 2009. OpenMP API for parallel programming, version 3.0. Retrieved from http://openmp.org/wp.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.

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. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

T. Rompf and M. Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Generative Prog. and Component Engineering (GPCE'10)*. 127–136.

T. Rompf, A. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. 2013. Optimizing data structures in high-level programs. In *Principles of Programming Languages (POPL'13)*. 497–510.

Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA.

B. Su and K. Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Supercomputing (ICS'12)*. 353–364.

X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao. 2011. Optimizing SpMV for diagonal sparse matrices on GPU. In *Parallel Processing (ICPP'11)*. 492–501.

A. Venkat, M. Hall, and M. Strout. 2015. Loop and data transformations for sparse matrix code. In *Programming Language Design and Implementation (PLDI'15)*. 521–532.

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

R. Vuduc, J. Demmel, and K. Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *J. Phys. Conf. Series* 16, 1 (2005), 521.

C. Whaley, A. Petitet, and J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 12 (2001), 3–35.

J. Willcock and A. Lumsdaine. 2006. Accelerating sparse matrix computations via data compression. In *Supercomputing (ICS'06)*. 307–316.

S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. 2009. Optimization of sparse matrixvector multiplication on emerging multicore platforms. *Parallel Comput.* 35, 3 (2009), 178–194.