RUNTIME SPECIALIZATION AND AUTOTUNING OF SPARSE MATRIX-VECTOR MULTIPLICATION

A Dissertation

by

Buse Yılmaz

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in the
Department of Computer Science

Özyeğin University
December 2015

RUNTIME SPECIALIZATION AND AUTOTUNING OF SPARSE MATRIX-VECTOR MULTIPLICATION

Approved by:

Asst. Prof. Barış Aktemur (Advisor)
Department of Computer Science
Özyeğin University

Research Assoc. Prof. María Garzarán
Department of Computer Science
University of Illinois at Urbana-Champaign

Manager, Scalable Runtimes
Intel Corporation

Asst. Prof. Hasan Sözer
Department of Computer Science
Özyeğin University

Asst. Prof. Kamer Kaya
Department of Computer Science and Engineering
Sabancı University

Date Approved: 24 December 2015

Assoc. Prof. Fatih Uğurdağ
Department of Electrical and Electronics Engineering
Özyeğin University

The only way to truly escape the mundane is for you to constantly be evolving, whether you choose to aim high or low.

— *Orihara Izaya*

ABSTRACT

Runtime specialization is used for optimizing programs based on partial information available only at runtime. In this thesis, we present a purpose-built compiler to quickly specialize Sparse Matrix-Vector Multiplication code for a particular matrix at runtime. There are several specialization methods and the best one depends both on the matrix and the platform. To avoid having to generate all the specialization variations, we use an autotuning approach to predict the best specializer for a given matrix. To this end, we define a set of matrix features for autotuning. Several of these features are unique to our work. We evaluate our system on two machines and show that our approach predicts either the best or the second best method in 91-96% of the matrices. 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 total runtime costs of autotuning and specialization can be amortized to bring performance benefits for many real-world cases.

ÖZETÇE

Koşut zamanda özelleştirme, sadece koşut zamanda belli olan kısmi veriye dayanarak programları optimize etmek için kullanılan bir yöntemdir. Bu tezde, seyrek matris-vektör çarpımı için hızlı bir şekilde koşut zamanda özelleştirme yapma amacına yönelik bir derleyici sunuyoruz. Seyrek matris-vektör çarpımı için çeşitli özelleştirme metotları vardır; en iyi yöntemin hangisi olduğu hem matris hem de donanım mimarisine bağlıdır. Özelleştirme yöntemlerinin tümünü kullanarak kod üretmekten kaçınmak için, otomatik ayarlama yaklaşımı kullanarak, girdi olarak verilen matris için en iyi özelleştiriciyi tahmin eden bir yöntem oluşturduk. Otomatik ayarlama yapabilmek için bir matris özellikleri kümesi tanımladık. Bu özelliklerin pek çoğu bizim çalışmamıza özgüdür. Sistemimizi iki ayrı makina üzerinde test ettik ve yaklaşımımız en iyi veya en iyi ikinci özelleştirme metotunu %91-96 oranında başarıyla tahmin edebilmektedir. Otomatik ayarlamayla yapılan tahminlerimiz, yalnızca en iyi metodlar kullanıldığında elde edilen hızlanmaya çok yakın hızlanmalar elde etmektedir. Verimli bir kod üreticisi ve dikkatlice oluşturulmuş bir matris özellikleri kümesi kullanarak, otomatik ayarlama ve özelleştirme süreçlerinin toplam koşut zaman masraflarını amortize edilebildiğini ve birçok gerçek-dünya matrisi için performans iyileştirmesi sağlanabileceğini gösterdik.

ACKNOWLEDGMENTS

I cannot thank my advisor Asst. Prof. Barış Aktemur enough. I deeply appreciate his guidance, his patience and his support. I gained an invaluable experience thanks to him. He has always been there to help me out to extend my knowledge and sharpen my academic abilities and, he guided me through my tough times. I am grateful for all his efforts to support me both academically and financially. I had a great time working with him and learning from him.

I am indebted to Prof. Sam Kamin, not just for his support and guidance throughout my research but also for his kind hospitality during my short stay in University of Illinois at Urbana-Champaign. I am also grateful to Prof. Grigore Roşu for inviting me to SSPL'12. It was a great two weeks full of theory and fun. Also, he met me several times during my stay in UIUC. I deeply appreciate for Norma Teyze's (Linton) friendship and hospitality. She took great care of me in Illinois. It would be a dull place without her.

I would like to express my gratitude for valuable feedback and guidance of my committee members; Asst. Prof. Hasan Sözer, Assoc. Prof. Fatih Uğurdağ, Asst. Prof. Kamer Kaya and Research Assoc. Prof. María Garzarán. Asst. Prof. Hasan Sözer and Assoc. Prof. Fatih Uğurdağ always spared time for discussions and guided me in thesis tracking meetings. Asst. Prof. Kamer Kaya was nice to observe my thesis progress and give valuable advice. I am indebted to Prof. Garzarán for her efforts and guidance throughout my research. Especially, she provided invaluable feedback and advices on the paper. I am happy to be a part of the joint research between Özyegin University and UIUC.

I am sincerely grateful to Asst. Prof. Furkan Kıraç for all his effort to guide me

throughout my research. I learnt a lot from him. His sharp comments and advices played an important role in making progress in my research.

I would like to thank my friends Ümit Akgün and Deniz Sökmen for their help in implementing some parts of our code generator.

Thanks to my lab mates Athar Khodabakhsh, Yunus Kılıç, Erkan Kemal and Gerard Djengomemgoto, my days in the lab were never boring. Also, I would like to thank Ali Arsal and Gönül Aycı for their valuable friendship. Lunches and coffee breaks could not get more enjoyable. I am thankful to my friends Barış Özcan and Murat Kırtay for being great TAs, it was a pleasure to work with you guys.

Last but not the least, I would like to thank my mom. I am indebted to her for always being there whenever I needed her and for supporting me. Without her love, support and patience, I would not be able to make it. This dissertation is dedicated to her.

I appreciate TÜBİTAK for granting me scholarship 2211 - National Scholarship Programme for PhD Students.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

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 sound, heat, electro dynamics, or quantum mechanics. They can also be obtained from graphs, in which case they represent the internet structure or social interactions.

Various iterative methods, such as Krylov subspace methods, exist. Usually, they converge after a large number of iterations. Thus, they are often combined with matrix preconditioners to decrease the number of iterations. Preconditioning can increase the runtime of each iteration, but the total runtime is reduced. The problem with preconditioning is that finding a good one is usually viewed as a combination of art and science [2]. For some matrices, there is simply no good preconditioner. Thus, the problem of generating efficient code for SpMV, the kernel operation in these iterative solvers, is a critical problem and one that has been and continues to be extensively researched [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].

Problems regarding generating efficient code for SpMV are listed below:

- The input matrix comes in various sparsity patterns. The SpMV kernel can be specialized for the sparsity pattern of the input matrix to obtain efficient code. Specialization can be performed offline for many problem domains (e.g. when the matrix, or at least its pattern, is known beforehand). If the matrix information is available only at runtime, online specialization can be performed.

Runtime specialization can pay off in iterative methods if the associated runtime costs are low enough; that is, if the specialized code can be generated quickly.

- There is no single best specialization method for all matrices. Several specialization methods should be developed. Their performance will be varying across input matrices and architectures. The best one for the current input matrix and architecture should be predicted without having to generate and run all the code variants. This can be done with autotuning.

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

- We investigate how accurately we can predict the best SpMV method for a given matrix. Our approach uses a Support Vector Machine (SVM) machine-learning technique to predict the best among 6 methods (including Intel's MKL as the baseline).

- For the SVM to predict the best specialization method, it is important to provide the set of features that determine the performance of SpMV. In addition to using matrix features that were previously used in other work, we list features that 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 in practical applications of iterative solvers 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 Chapter 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.

In the rest of this chapter, we explain the problem of SpMV, related performance issues, and the runtime specialization approach to SpMV.

1.1 SpMV and Performance Issues

Sparse matrix vector multiplication is the operation

$$w \leftarrow w + A \cdot v$$

where a sparse matrix A is multiplied with a dense vector v, and the result is stored in vector w, which is also dense. Although today's modern microprocessors exhibit astounding computational power, to achieve close-to-peak FLOP performance, the CPU has to be fed with data continuously. As new chips are developed, ratio of peak memory bandwidth to peak FLOP ratio is decreasing with the increase in core counts, further limiting the performance of bandwidth-limited applications [14, 6]. SpMV is notorious for being memory-bound and obtaining only small fractions of the peak performance on modern microprocessors [14, 6]. While dense matrices are stored in two dimensional arrays, custom representations are used for sparse matrices to reduce space requirements. The sparse representation is a major factor in utilizing the CPU. Perhaps the most popular sparse matrix representation is the *Compressed Sparse Row* (CSR) format. An example that illustrates CSR format is given in Figure 1.

In CSR format, the matrix is represented with three arrays, namely, *vals*, *cols*

and *rows*. *vals* array holds the nonzero elements of the matrix. *cols* array holds the column indices of nonzero elements, and finally, in the *rows* array, indices of the first nonzero element of each row are stored.

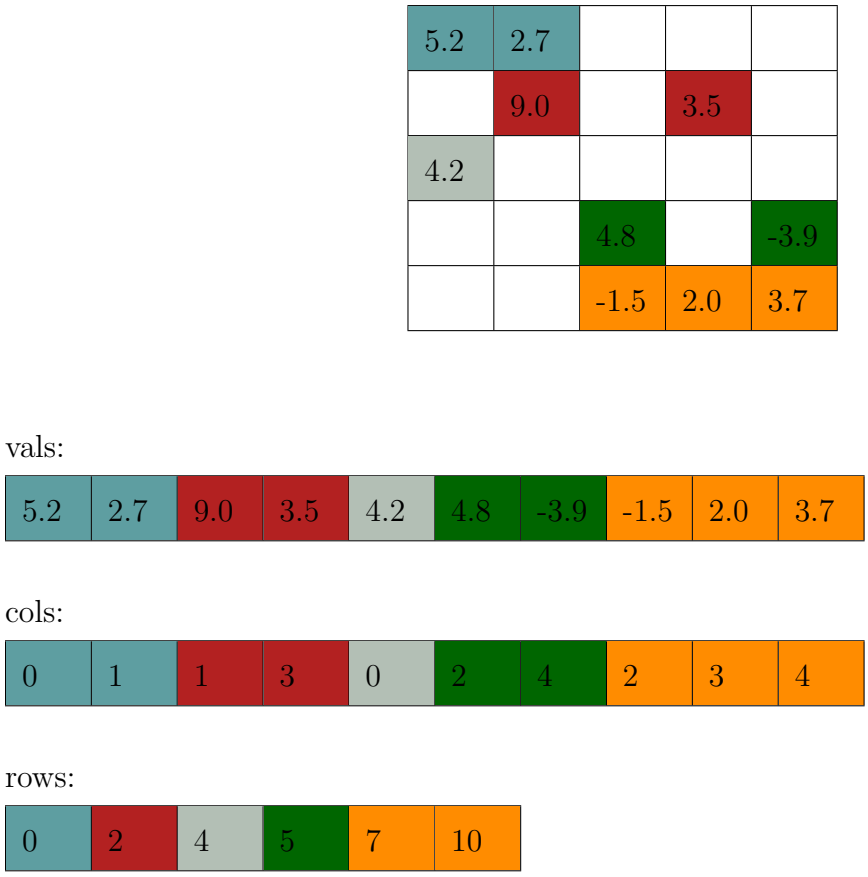


Figure 1: CSR Format Example.

Figure 2 shows the SpMV operation using the CSR format. The given code is in C syntax. While providing reduction in space requirements, storage formats like CSR introduce indirect references (as in `v[cols[j]]`). These references also cause irregular access patterns (in the input vector `v` for CSR). Irregular memory accesses reduce cache utilization. Also, many matrices exhibit a large number of rows with short length, causing a degradation in instruction-level parallelism (*ILP*) [15]. Hence,

sparsity regime of the input matrix introduces various challenges. There is no silver-bullet format or multiplication algorithm optimizing SpMV for every matrix and platform. Autotuning is an effective approach to pick the most appropriate storage format for a given matrix on a particular platform [16, 17, 18, 19, 20, 21, 22]. Here, we use autotuning to select the best specialization method for a given matrix.

```
// w <- w + A.v, where A is in CSR
for (int i = 0;  i < rowCount; i++) {
  double s = w[i];
  for (int j = rows[i];  j < rows[i+1]; j++) {
    s += vals[j] * v[cols[j]];
  }
  w[i] += s;
}
```

Figure 2: SpMV implementation with CSR

Challenges regarding the performance of SpMV is studied extensively by Goumas et al. [15]; below is a summary of the key points:

- Memory intensity (Lack of temporal locality): Kernel is memory bound.
- Indirect memory references: A sparse storage format requires storing indices of nonzero values in separate data structures. This implies additional load operations, indirect memory accesses (which is bad for the CPU pipeline), traffic for the memory subsystem, and cache interference.
- Irregular access patterns for vector v: Access to v is dependent on the sparsity regime of the matrix and is irregular in general.

- Short row lengths: Many sparse matrices exhibit a large number of rows with short length. This may hurt the performance due to loop overheads that become significant with short loop body.

1.2 Optimizations on SpMV

To optimize SpMV and address performance bottlenecks mentioned in Section 1.1, many techniques have been proposed.

- **Reducing bandwidth requirements:** Using efficient storage formats to store only nonzeros of the matrix along with their indices is the most common approach. Examples are BiELL [23], BCOO [24], BRC [25], BELLPACK [26], BTJDS [27], CSB [5], CSX [7], CSR-DU and CSR-VI [28], ELL-R [29], ELLPACK-RT [30], ELL-RT [31], Hybrid ELLPACK/COO (ELL, COO, HYB) [11], PBR [32], RowPattern CSR (RPCSR) [33], Cocktail Format [34]. A more complete list of storage formats is provided in [35].

  In addition to storage formats, index data reduction is a technique to reduce bandwidth requirements [36, 33, 37, 32]. A problem with this approach is that padding with zero may be necessary depending on the matrix structure [7]. An overview of blocking storage formats and performance issues of blocking is provided in [38]. Reordering techniques to improve locality and minimize communication costs are studied in [39, 40, 41, 42]. Williams et al. provide a detailed study of these techniques on several multicore architectures [8].

- **Irregular access patterns in vector v and indirect indexing:** Irregular access patterns in the input vector v introduce bad cache behavior due to poor data locality. Permutation of the matrix or column reordering in favor of cache reuse is known to be an effective technique. Im et al. [43] state that the performance of sparse matrix operations tends to be much lower than their

dense matrix counterparts for two reasons: (1) the overhead of accessing the index information in the matrix structure, (2) the memory accesses tend to have little spatial or temporal locality. They provide the following case study: On an 167 MHz UltraSPARC I, there is a 2x slowdown due to the data structure overhead (measured by comparing a dense matrix in sparse and dense format) and an additional 5x slowdown for matrices that have a nearly random nonzero structure.

Irregular access patterns and indirect memory access problems are usually addressed using matrix reordering, register blocking and cache blocking [43, 44, 45, 46, 47]. Toledo et al. address irregular access patterns to vector v by applying blocking to exploit temporal locality and to reduce indirect indexing [45]. Vuduc et al. extend the work in [43], but focus on register blocking only [46]. Pinar et al. show that permuting the nonzeros is NP-Complete and propose a graph model to reduce it to Traveling Salesman Problem [44]. Vuduc et al. apply blocking and split the matrix into a sum and store each submatrix in their own format UBCSR [47]. Temam et al. do an analysis of cache behavior of SpMV, and point out the problem of irregular access in [48].

Reordering may improve the locality of accesses to vector v, but the accesses to the matrix data and output vector w may no longer be regular after the reordering.

- **Short row lengths:** Mellor-Crummey and Garvin, and White et al. address performance issues related to large number of rows with small lengths in [10, 49]. White et al. [49] state that in addition to data locality, rows with small lengths, which are frequently encountered in sparse matrices, can drastically hurt the performance (due to ILP reduction).

1.3 Runtime Code Generation and Specialization

Specializing a general-purpose program to a specific context provides efficiency gains. However, writing a special-purpose program needs more effort and time investment as well as domain-specific knowledge. To compensate, one can use a *code generation* approach to write a general-purpose program that produces specialized code at runtime.

Code generation (aka *specialization*) is the process of writing programs that write programs. The main idea is to reduce human errors, improve efficiency, productivity, modularity and customization. There are various ways to do code generation: Macros, ad-hoc code generation with strings, quasi-quotations, etc. Code generation provides one with the ability to produce specialized code by utilizing domain-specific knowledge. Code generation can be done at compile-time or at runtime depending on when the inputs become available. If performed at runtime, code generation is beneficial when the generated code is to be used many times. This is due to the fact that code generation process introduces a runtime cost. There is a break-even point indicating when the generated code compensates for its cost, as shown in Figure 3. Iterative solutions are appropriate for runtime code generation because they involve SpMV of a particular matrix with many vectors.

One of the earliest examples of code generation is by Ken Thompson [50]. Runtime code generation is the key technology behind just-in-time(JIT) compilers, compiling interpreters [51].

Notable performance improvements has been achieved due to specialization in the areas of operating systems [52, 53, 54], method dispatch in object-oriented systems [55, 56], fast dynamic code generation system [51], compiler extensions using staging [57], eliminating abstraction overhead from generic programs using multi-stage programming [58], and building efficient query engines using generative programming in high-level languages [59].

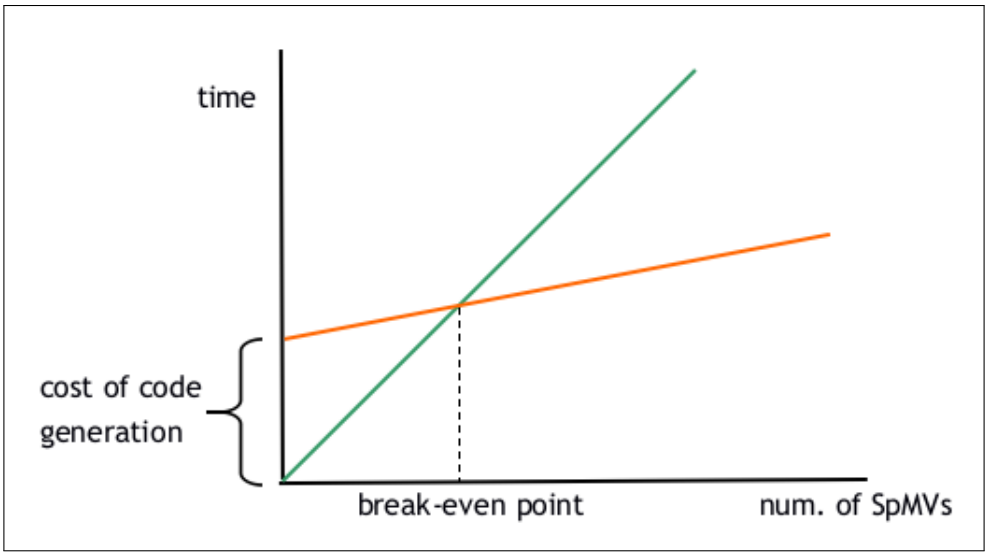


Figure 3: Program generation is beneficial when the resulting code compensates for the code generation cost.

1.4 Problem Statement and the Solution Approach

SpMV is a crucial operation in scientific computation. In many contexts (e.g. in iterative solvers), a fixed sparse matrix is multiplied with several different vectors. Hence, SpMV is an appropriate problem to apply code generation: the code can be specialized to the matrix in hand once, and then used many times.

Problem:

There are various methods for SpMV specialization. It has been shown that a specialization method provides substantial speedup, however, no single method is the best; the best method varies across machines and across matrices [1].

Solution Approach:

We use autotuning to predict the specialization method that will yield the best performance for a particular matrix on a particular machine. This way, we can avoid generating and profiling the performance of all the code variants. In doing so, we pay extra care that the runtime costs associated with autotuning (i.e. analysis of the matrix for prediction, and code generation for the predicted method) are low enough,

so that performance benefits are obtained. To this end, we developed an SpMV specialization library that rapidly generates code at runtime, and an autotuning system for the SpMV specialization methods we are using.

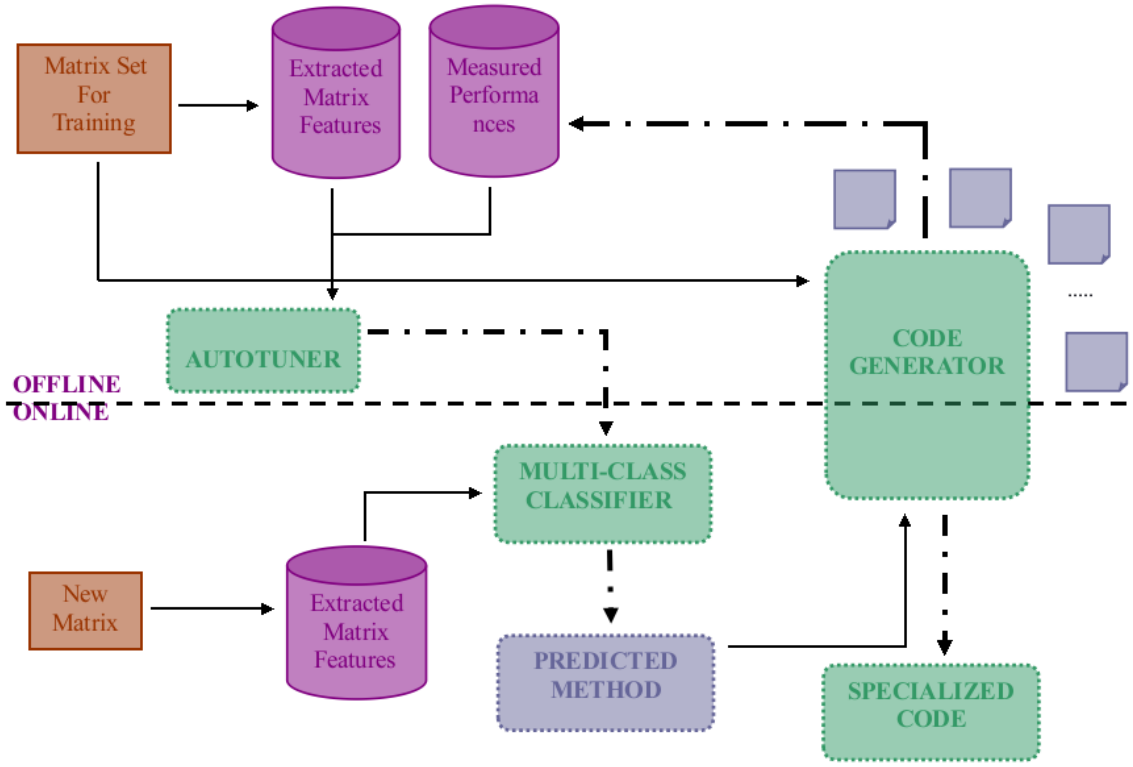


Figure 4: Runtime specialization library (SpMVLib) and the autotuner predicting the best specializer and generating its code, given information from install time.

Our library is depicted in Figure 4. It combines a multi-class classifier with a runtime code generator. Given an input matrix at runtime, it predicts the best specializer to generate based on offline training. It generates specialized code for the input matrix at runtime.

The autotuner is a hybridization of offline benchmarking and performance modeling at run-time. At install time, predefined matrix features are collected. Then, for each matrix and each specializer, code is generated, and the performance results are collected. This information is used to train the multi-class classifier. At runtime, when a new matrix is provided, the autotuner predicts the best specialization method to be used based on its knowledge from install time. This general autotuning

approach has been successfully used in prior art [60, 61, 22, 62, 63, 64, 65, 66].

Either a specialization method or the nongenerative baseline method can be predicted by the autotuner. The multi-class classifier is based on a machine learning model (we use a Support Vector Machine). For a new input matrix, given the prediction, the SpMV Library generates specialized code for the matrix or if the baseline method is predicted, it runs the baseline method (since baseline method is nongenerative).

1.5 Organization of the Dissertation

This dissertation is organized as follows. The next Chapter describes specialization methods used in the library. In Chapter 3, our code generation library is explained in detail. This is followed by Chapter 4 where the autotuning framework is explained in detail. In Chapter 5, we present our experimental setup and provide experimental results. In Chapter 6, we evaluate the latency incurred by runtime prediction and code generation. In Chapter 7, we discuss other optimizations considered for the code generation library. Code generation costs and break-even points are provided. Chapter 8 is where related work for SpMV and code generation is presented. This is followed in Chapter 9 by our conclusions.

CHAPTER II

SPMV SPECIALIZATION METHODS

In this chapter, we describe the methods that can be used to specialize the SpMV code. In the discussion of the methods below, we assume A is an N×N matrix, with NZ nonzeros. We use C notation in the code snippets below. `rows` array contains the row indices, `cols` array contains the column indices, and `vals` array contains the nonzero elements of the matrix. The type of `rows` and `cols` is `int*`, and `vals` is `double*`; `v` is the input vector, `w` is the output vector. Also, in ***Data order***, ***Data size*** and ***Code size***, "nz" is number of nonzeros, "n" is number of rows, "ner" is the number of non-empty rows.

As mentioned in Section 1.1, in the CSR format for sparse matrices, the `vals` array contains NZ double precision floating-point values; the `cols` array contains the column indices of nonzero elements (NZ integers); and the `rows` array contains, for each row, the starting/ending index of elements in the `rows` and `vals` arrays (N+1 integers). Hence,

Data order: CSR does not reorder the data.

Data size: Data size is equal to $nz * 8 + nz * 4 + (n + 1) * 4$.

Code size: CSR's code is given in Section 1.1, Figure 2. It has one for-loop for each row. We assume constant code size c for CSR.

Each specialization method imposes a custom layout of the matrix data. Therefore, the interpretation and size of the arrays change according to the method.

2.1 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 [10]. 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 an important factor in SpMV performance [15]. Code that would be generated by *CSRbyNZ* for 100 rows with a length of 3 is given in Figure 5.

```
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]];
}
// Set the pointers for the next loop.
rows += 100;
cols += 100*3;
vals += 100*3;
```

Figure 5: Sample code for CSRbyNZ

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 the CSR format, except for when there are rows with

no elements. Data size is given as:

$$nz * 8 + nz * 4 + ner * 4$$

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. Code size is given as:

$$\sum_{i=1}^{Row\_nz} nz\_row_i * c_1 + Row\_nz * c_2$$

where "Row_nz" is the number of distinct row lengths and "nz_row_i" is the number of nonzeros is group i.

2.2 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 $r+3$, 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 in Figure 6.

Data order: *RowPattern* reorders the matrix data to group rows with the same pattern together; similar 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. Similar to CSRbyNZ, the length of the `rows` array is equal to the number of nonempty rows.

```
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];
}
// Set the pointers for the next loop.
rows += 100;
vals += 100*4;
```

Figure 6: Sample code for RowPattern

Data size is given as:

$$nz * 8 + ner * 4$$

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. If a pattern is unique to only one row, completely unfolded code is generated. We distinguish these cases in the formula below.

Code size is given as:

$$\sum_{i=1}^{rowPatterns\_single} c_1 + \sum_{i=1}^{rowPatterns\_multi} nz\_rowPattern_i * c_2 +$$

$$(rowPatterns\_single + rowPatterns\_multi) * c_3$$

where "$rowPatterns\_single$" and "$rowPatterns\_multi$" are number of distinct row patterns for patterns that cover single and multi rows. "$nz\_rowPattern_i$" are the number of rows in row pattern group i.

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 [15].

2.3 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 [9]. A motivation of this method is to avoid the zero-fill problem of OSKI [18] that generates efficient per-block code by inserting some zeros into the matrix data. *GenOSKI* generates one loop for each block pattern of nonzeros in the matrix. 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, in this case, store indices of *blocks*, not individual nonzero elements. The index of a block is the location of the top-left corner of the block.

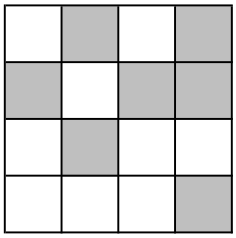
Data order: *GenOSKI* reorders matrix data to group blocks with the same block pattern together. The accesses to the output vector `w` are sequential within a block, but not across blocks.

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.

Data size is given as:

$$nz * 8 + nblocks * (4 + 4)$$

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.



```
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];
}
// Set the pointers for the next loop.
rows += 100;
cols += 100;
vals += 100*7;
```

Code size is given as:

$$\sum_{i=1}^{patterns} nz\_pattern_i * c_1 + patterns * c_2$$

where "$patterns$" is the number of block patterns and "$nz\_pattern_i$" is the number of blocks with block pattern i.

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

For the evaluation in this studys 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 ***GenOSKI*55**, respectively.

2.4 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, etc. — 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. To give up-front information, we have measured *Unfolding* as the best method for 13-21 matrices out of 610. For these matrices, *Unfolding*'s performance was on the average 1.23× to 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. Because the

nonzero values of the matrix are available, this is a valid optimization. Hence, the statements are compiled as if the code were

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

Because the matrix values are loaded from a constant pool, they can be loaded to registers once and reused multiple times, similar to

```
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. [28] as "Value Compression". We also reported the impact of unique values on the performance in [1] by Kamin et al.

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`]`. Furthermore, the inverse of distribution of multiplication over addition can be performed. E.g. `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.

Data order: Data order is kept as the original.

Data size: Since *Unfolding* stores only distinct nonzeros, if number of nonzeros are less than 5000, data size is reduced for these matrices. Else, all nonzeros are stored. Data size is given as:

$distinct\_nz * 8$

Code size: *Unfolding* simply unrolls loops, hence, code size is likely be proportional to the number of nonzeros. However, the optimizations we discussed previously reduces the code size.

Code size is given as:

(possibly) $nz * c$

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, 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. We implemented a version of CSE and performed several experiments with it (details are in Section 7.2), but we did not incorporate CSE into our final code generator.

To give concrete evidence of the impact of *Unfolding* optimizations, let us look at Table 1. 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*,

Matrix		Unique	MUL inst.	Memory traffic (MB) and Speedup wrt Baseline					
N	NZ	values		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×

Table 1: The impact of optimizations possible in Unfolding. Best performing method's speedup is in **bold** font.

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 [14], 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 (one for read, one 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 [1], we presented formulas to calculate the matrix data and code sizes for these methods.

We show information for 9 matrices in Table 1. *Unfolding* gives the best performance for the first 7 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 5 matrices have few unique elements while the other 4 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 1 are two such matrices. Even though the optimizations we discussed above are not applicable to these matrices, *Unfolding* performs very good 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 [67], DIA [2], SKS [36], etc.) 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 fix our set of methods at some point and we specifically focus on specialization methods, not generic storage

formats. (Keep in mind that specialization may be applied to those other formats as well.) That said, in work by Kamin et al. [1], we had compared the specialization methods that we evaluate in this work with BiCSB [6] and CSX [7]. The specialization methods we use in this work had performed the best most of the time. We had also experimented with hybrid approaches, but had not obtained high speedups. Hence, in this work, we limited ourselves to the five specialization methods and a baseline implementation.

CHAPTER III

CODE GENERATOR

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 [68, 69]. 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, global value numbering, etc.). 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 equal number of nonzero values to each partition, using the following approach: If there are t threads, starting from the first row, we assign consecutive 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 [8, 9, 70, 13]. For each partition, a function is generated using the specified specialization method. The generated functions are

ALGORITHM 1: The pseudo-code of the CSRbyNZ code generator. This generator produces X86_64 code for each distinct row length in a matrix, corresponding to the source snippet on page 13.

```
// 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**
    $M \leftarrow$ *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
    $P \leftarrow$ *current position in the object code buffer*;
    emit(xor %xmm0, %xmm0) ; // reset xmm0 to 0
    for $i \leftarrow 0$ **to** L **do**
        // Emit code to calculate vals[b+i]*v[cols[b+i]] and accumulate in %xmm0
        emit(mov $i \times 8$(%r8,%r9,8), %xmm1) ; // xmm1 ← vals[b+i]
        emit(mov $i \times 4$(%rcx,%r9,4), %rax) ; // rax ← cols[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 $M \times 4$, %rdx) ; // rows ← rows + M
    emit(add $M \times L \times 4$, %rcx) ; // cols ← cols + $M \times L$
    emit(add $M \times L \times 8$, %r8) ; // vals ← vals + $M \times L$
end

executed concurrently using OpenMP [71]. 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 Chapter 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 can.

The way we generate assembly code is mostly straightforward. Algorithm 1 provides the *CSRbyNZ* code generator in pseudo-code. This generator produces X86_64 machine code corresponding to the sample source code given for *CSRbyNZ* in Section 2.1. The X86_64 code generated by this *CSRbyNZ* generator for the t2em matrix is shown in Figure 7.

We wrote our own *emit* functions to write specific bits into the in-memory object code buffer for the given opcode and arguments. A sample *emit* function, *emitRegInst*, is provided in Figure 8. This function handles emission of several register-to-register

```
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
; continued on the right
```

```
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
```

Figure 7: 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.

```
void SpMVCodeEmitter::emitRegInst(unsigned opCode, int XMMfrom, int XMMto) {
  unsigned char data[5];
  unsigned char *dataPtr = data;
  if (opCode == X86::ADDPDrr)
    *(dataPtr++) = 0x66;
  else if (opCode == X86::ADDSDrr || opCode == X86::MULSDrr || opCode == X86::SUBSDrr)
    *(dataPtr++) = 0xf2;

  if (XMMfrom >= 8 && XMMto < 8) {
    *(dataPtr++) = 0x41;
  } else if (XMMfrom < 8 && XMMto >= 8) {
    *(dataPtr++) = 0x44;
  } else if (XMMfrom >= 8 && XMMto >= 8) {
    *(dataPtr++) = 0x45;
  }
  *(dataPtr++) = 0x0f;

  switch (opCode) {
  case X86::ADDPDrr:
  case X86::ADDSDrr: *(dataPtr++) = 0x58; break;
  case X86::SUBSDrr: *(dataPtr++) = 0x5c; break;
  case X86::XORPSrr: *(dataPtr++) = 0x57; break;
  case X86::FsMOVAPSrr: *(dataPtr++) = 0x28; break;
  case X86::MULSDrr: *(dataPtr++) = 0x59; break;
  default:
    std::cerr << "Unsupported opcode.";
    exit(1);
  }

  unsigned char regNumber = 0xc0 + (XMMfrom % 8) + (XMMto % 8) * 8;
  *(dataPtr++) = regNumber;

  DFOS->append(data, dataPtr);
}
```

Figure 8: Our emitting function that emits various register-to-register instructions.

instructions (`ADDPDrr`, `ADDSDrr`, `MULSDrr`, `SUBSDrr`, `XORPSrr`, and `FsMOVAPSrr`). We wrote emit functions by examining the bits corresponding to instruction opcodes and their arguments as output by compilers.

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 [1]. We compiled these codes using icc with flags `-O3 -no-vec` (vectorization disabled, because our

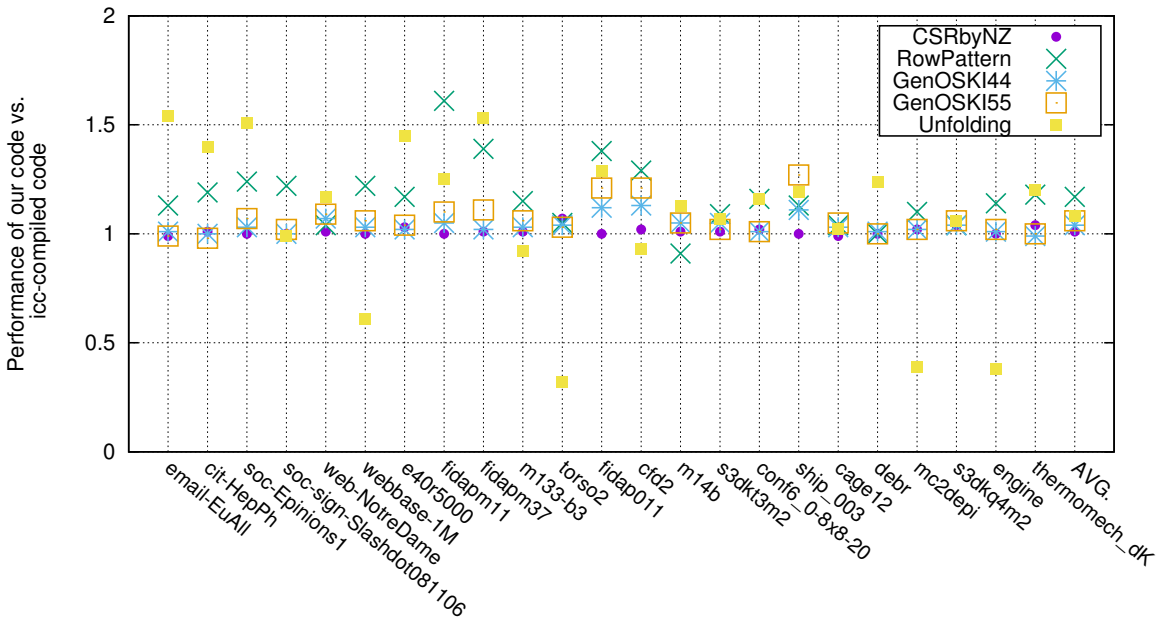


Figure 9: The performance ratio of our compiler's output to icc's output for the matrices used in [1]. A value greater than 1 means we generated more efficient code than icc.

generator does not do vectorization). We measured the performance of the compiled code and compared against our code generator.

In Figure 9, 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, smaller than 1 means icc's output performed better. The test was done on our Intel CPU testbed machine using single-threaded execution. For *CSRbyNZ*, *GenOSKI44*, and *GenOSKI55,* the ratio is consistently close and slightly above 1. On the average (last column in Figure 9) ratios are, 1.01 for *CSRbyNZ*, 1.04 for *GenOSKI44*, and 1.06 for *GenOSKI55*. For *RowPattern*, our code performs better than icc for 21 cases out of 23. On the average, the 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 the average.

Table 2 shows the best of the 5 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 20 matrices out of 23, the winner method both for icc-compiled code and our compiler is the same. These codes perform similarly, with our compiler's output giving 1.04× the performance of icc. The overall performance ratio of our code to icc is 0.99×. The matrices for which winner methods differ are indicated in bold. For webbase-1M, the winner method when using our generator is *CSRbyNZ*, while it is *Unfolding* with icc. Similarly, for mc2depi, it is *RowPattern* vs. *Unfolding* and for fidapm37, it it *GenOSKI55* vs. *GenOSKI44*. While the performance gap between our generator and icc is large for webbase-1M (23%), performances for mc2depi and fidapm37 are close.

There are 4 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.

Another optimization that icc applies over *Unfolding* is very similar to our *RowPattern* specializer. *RowPattern* finds the exact pattern of nonzero entries in each row and generates a loop for each pattern. Similarly, icc detects memory access patterns of rows and generates a loop for them. Due to their sparsity pattern, mc2depi and

Matrix	Best performing method when using		our code / icc
	our generator	icc	
email-EuAll	CSRbyNZ	CSRbyNZ	0.99
cit-HepPh	CSRbyNZ	CSRbyNZ	1.01
soc-Epinions1	CSRbyNZ	CSRbyNZ	1.00
soc-sign-Slashdot081106	Unfolding	Unfolding	0.99
e40r5000	RowPattern	RowPattern	1.17
fidapm11	CSRbyNZ	CSRbyNZ	1.00
m133-b3	CSRbyNZ	CSRbyNZ	1.01
torso2	RowPattern	RowPattern	1.05
fidap011	GenOSKI44	GenOSKI44	1.12
cfd2	CSRbyNZ	CSRbyNZ	1.02
m14b	CSRbyNZ	CSRbyNZ	1.01
s3dkt3m2	RowPattern	RowPattern	1.09
conf6_0-8x8-20	RowPattern	RowPattern	1.16
ship_003	CSRbyNZ	CSRbyNZ	1.00
cage12	CSRbyNZ	CSRbyNZ	0.99
debr	CSRbyNZ	CSRbyNZ	1.00
s3dkq4m2	RowPattern	RowPattern	1.04
engine	Unfolding	Unfolding	0.38
thermomech_dK	GenOSKI44	GenOSKI44	0.99
web-NotreDame	GenOSKI44	GenOSKI44	1.07
mc2depi	**RowPattern**	**Unfolding**	**0.94**
webbase-1M	**CSRbyNZ**	**Unfolding**	**0.77**
fidapm37	**GenOSKI55**	**GenOSKI44**	**1.07**
Avg.			**0.99**

Table 2: Comparing the performance of the code compiled by icc with our code generator.

torso2 benefit from this optimization. This results in aggressive optimization over *Unfolding* since loop shortens the code and compensates for lengthy code generated by *Unfolding*. We do not apply this optimization in our *Unfolding* specializer, because we already have the *RowPattern* method. It is not surprising that among our methods, *RowPattern* performed the best for torso2 and mc2depi.

In Figure 10, there is a small portion of the generated code using *Unfolding* with icc for torso2. We reduced the precision here due to space limitations. As seen in the code snippet, except for the first row, 3 consecutive rows have the same index values for accessing vector `v`, differing only with an offset; e.g. for the third row we add 1

```
w[0] += .86 * v[0] + -.06 * v[1] + -1.15 * v[65] + .28 * v[130] + .06 * v[103870];
w[1] += .06 * v[0] + .86 * v[1] + -.06 * v[2] + -1.15 * v[66] + .28 * v[131];
w[2] += .06 * v[1] + .86 * v[2] + -.06 * v[3] + -1.15 * v[67] + .28 * v[132];
w[3] += .06 * v[2] + .86 * v[3] + -.06 * v[4] + -1.15 * v[68] + .28 * v[133];
```

Figure 10: Code generated using Unfolding with icc for torso2.

to the second row's indices. icc detects this and generates a loop for rows with such access pattern. The reflection of this optimization over performance is provided in Table 3. Although *RowPattern* performances of both our generator and icc are close to each other, this optimization of icc results in *Unfolding* to perform very close to *RowPattern* for icc. Yet, *Unfolding*, beating *RowPattern*, became the best method for mc2depi.

	Our generator		icc		$icc_{Unfolding}/$	Winning method		
Matrix	RowPattern	Unfolding	RowPattern	Unfolding	$our_{Unfolding}$	ours	icc's	Winners ratio
torso2	1000.42	3640.25	1046.824	1148.856	0.32	Row Pattern	Row Pattern	1.05
mc2depi	3390.77	8188.56	3713.255	3196.843	0.39	Row Pattern	Unfolding	0.94

Table 3: Performance comparison of RowPattern and Unfolding specializers for both our code generator and icc. Runtimes provided are per iteration.

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 to take at runtime. To give a measure, compiling the C source codes for 23 matrices took about two days on our testbed machine. Code generation has to be very rapid for runtime specialization to pay off. That is the reason why we wrote our purpose-built compiler.

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. However, we separately experimented with vectorization and CSE. We report details of these experiments in Chapter 7.

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, 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. [72]) to satisfy dimension (2) and (3), but not (1); compiling templates separately misses inter-template optimization opportunities. We could have generated code at an AST or intermediate representation level (e.g. with Jumbo [73] or LMS [74]) and use an existing compiler back-end to optimize the generated program, but this would fail to satisfy dimension (2). We opted for dimension (1) and (2) at the price of (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.

CHAPTER IV

AUTOTUNING

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 [75, 61, 76, 22, 77, 66]. The same problem recurs in specialized SpMV code; the best performing SpMV specialization method depends on both the matrix and the machine [1]. In this chapter 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 non-trivial cost (we discuss code generation costs in Chapter 6).

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 non-generative one (i.e. Intel's MKL as the baseline) are executed and their performances are recorded.

2. The collected data are used to train a multi-class 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 multi-class 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 chapter we evaluate how one can accurately predict the best SpMV method for a particular matrix. Our experimental results (Chapter 5) 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 the memory [14]. Non-specialized 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 Chapter 2, we commented on the code and data sizes implied by each method. In [1], we used formulas to compute code and data size for the different methods. 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 (Chapter 5).

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

Table 4: Number of times a method yields the smallest size (code and data size).

In Table 4, we provide the number of times each method has the smallest size. *CSRbyNZ* is the smallest for only 63 times, but it performs the best for many more matrices (see Figure 12 in Chapter 5). 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 1 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 looking at 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 performances 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 5 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),

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)

Table 5: Matrix features grouped under the method they impact the most.

number of nonzeros (NZ), and nonzeros per row to represent the general structure of a matrix. We also include the number of nonempty rows 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; in these, 28 have more than 20% of their rows empty. From our point of view, 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 (resp. blocks) per pattern, and the average length of patterns indicate the average runtimes of generated loops. *RowPattern* generates a loop for each pattern; however, if a pattern is unique to only one row, a 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. [9]. 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. GenOSKI-related features are collected for both 4×4 and 5×5 block sizes.

Unfolding's performance is highly sensitive to the number of distinct NZ values as discussed in Section 2.4. 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* of 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. The number of rows and nonzeros of

the matrix are almost always collected as features. (e.g. [78, 79, 80, 81, 82]). Average NZ per row is also common [78, 34, 81]. Some other features used in the literature are

- zero-fill ratios for formats like DIA [36], ELL [67] and BELLPACK [26] in [83, 81],
- variation of row lengths [83, 79, 80, 81],
- mean neighbor count of nonzero elements [79, 80],
- number of blocks and dense blocks per super row [34],
- number of diagonals, number of nonzero elements per diagonal [34, 81],
- max number of nonzeros per row [81, 82], and
- memory traffic (number of bytes fetched, number of writes to `w`) [9].

In an attempt to give more information to the learner, we also experimented with other features as well. For instance, we decomposed the properties in the form of histograms to carry more fine-tuned information. E.g. the number of row patterns whose length is less than 3, between 3 and 10, and more than 10, etc. (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 over-fitting (a.k.a. the curse of high dimensionality).

4.2.1 Full vs. Capped Feature Set

We call the features listed in Table 5 the **full feature set**. In Chapter 5, we will see that full feature set gives us good prediction success, however, it is expensive to compute. As an alternative, we have an option to 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 feature set 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. But analysis continues normally for other features. The number of distinct values is always capped at 5000, in both full and capped feature extraction.

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 Chapter 5.

We performed a correlation analysis between the features, shown in Figure 11. 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, 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 (resp.

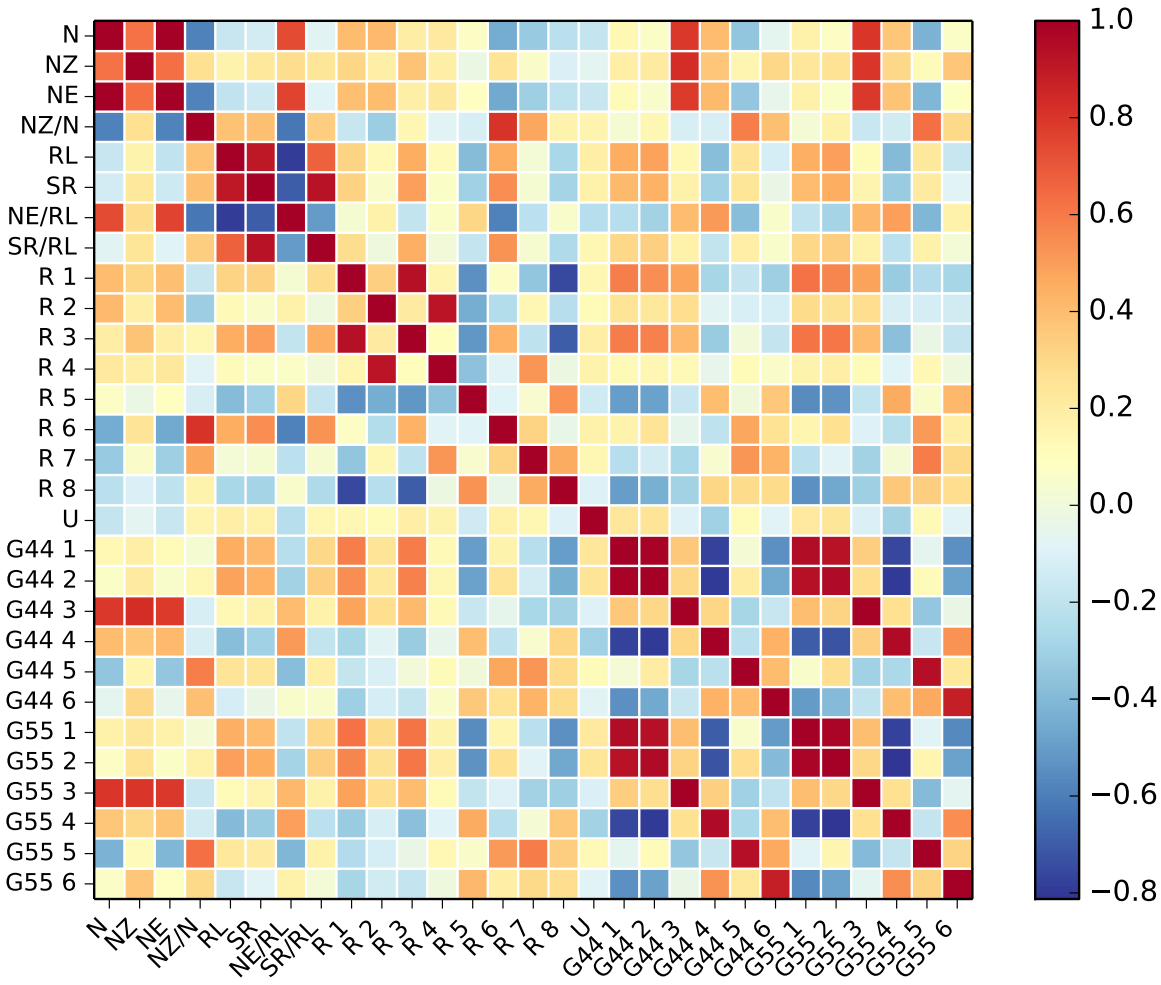


Figure 11: Correlations between features of the full feature set.

distinct row lengths) and the sum of pattern lengths (resp. sum of row lengths) in *RowPattern*, GenOSKI, and *CSRbyNZ* methods. This is also normal; the sum of pattern lengths increases as the 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 options to pick from as the learning model for multi-class classification. We experimented with many, including Random Forest Classifier and Decision Tree Classifier. We found C-Support Vector Classification (SVC) to give the best results when it is used with RBF (Gaussian) as the kernel. 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 and Labeling Approaches

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

4.4.1 Naive Labeling

We used the best performing method for a matrix as its class. In this approach, there are as many classes as SpMV methods (6 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× in our test setup. The ratio is less than 1.01× in 6-8% of the matrices, less than 1.02×

in 12-16%, less than 1.05× in 24-36%. We try to remedy this potential problem with the next approach.

4.4.2 Paired Labeling

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-MKL*, we generate code for *CSRbyNZ*. This is the best case for prediction. If the matrix is classified as, say, *RowPattern-Unfolding* or *RowPattern-CSRbyNZ*, we generate code using the *RowPattern* method. 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, 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. The next approach explains this.

4.4.3 Paired Labeling with a Threshold

If the best method is better than the second best method by more than the threshold, 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× of *Unfolding*, and the threshold value is 1.05×. 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.

The results of the naive and paired labeling approaches are presented in Chapter 5. Thresholding did not sufficiently improve the prediction results; we also give a brief discussion about this in Chapter 5.

Labeling happens automatically, with no human effort. For each matrix, the autotuner looks at the performance measurements of the SpMV methods, and determines the class using the chosen approach (naive or paired).

CHAPTER V

EXPERIMENTAL RESULTS

In this chapter, we provide the experimental results for both performance comparison and prediction success of the classifier.

We provide, for 5 specialization methods and the baseline method (Intel's MKL), the number of times each method becomes the best across our two testbeds and with different thread counts. We see that 86-94% of the time a specialization method becomes the best method. The results show that there is no best method for all machines and for all thread counts. We also observe differences in best method count for a specialization method when thread counts change. The average and maximum speedups for both machines and with different thread counts range between 1.33-1.83× and 2.94-5.69× respectively.

We discuss the matrix features that we selected and why these are important features. We also discuss and evaluate two different class labeling approaches that we selected to train the SVM. Overall, our experimental results using 610 matrices on 2 different machines show that in 71-86% of the matrices the best method can be predicted, in 11-20% of the matrices, the second best method can be predicted. Only 5-8% of the predictions choose a method worse than the baseline. Predicted methods give an average speedup of 1.31, 1.41, 1.37, and 1.77 on four scenarios, where the maximum achievable average speedups are 1.33, 1.45, 1.39, and 1.83, respectively.

5.1 Experimental Setup

In our experimental evaluation, we use a set of 610 matrices obtained from the Matrix Market [84] and the University of Florida collection [85]. All our matrices are square and sparse. Their number of nonzero elements range from 100K to 15M, dimensions

Name	Processor & Freq (GHz) (Microarchitecture)	Cores	Cache Sizes (Bytes)			Mem	Linux OS	compiler
			L1 (I/D)	L2	L3	(GB)		
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

Table 6: Target Platforms

range from 2K to 2.4M. 129 of the matrices 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 [5, 7, 8]. 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, etc.

We executed code on two unloaded X86_64 machines, one with an Intel (turing), the other with an AMD processor (milner). The properties of our testbed computers are in Table 6. On both machines we generated code using 5 specialization methods (CSRbyNZ, RowPattern, GenOSKI44, GenOSKI55, Unfolding). We also collected the runtime of Intel MKL's SpMV function, and we use MKL as the baseline when we calculate speedups. So, in total 6 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 2 seconds) 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 three 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 three 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 (6 on turing, 8 on milner). We refer to the sequential runs as **turing-1** and **milner-1**, parallel runs as **turing-6** and **milner-8**.

For comparing the performance of specialized code, we use Intel's MKL library as the baseline implementation. We could not use the AMD Core Math Library (ACML) [86] on our testbed machine that has an AMD CPU, because ACML does not have SpMV. ACML was recently retired and replaced by the BLIS [87] and libflame [88] libraries, but these library do not provide SpMV, either.

For prediction experiments, we used the scikit-learn module of Python (version 2.7.9) [89]. We applied 10-fold cross validation for training and testing. This is a standard approach in machine learning. We first shuffled the data, 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.

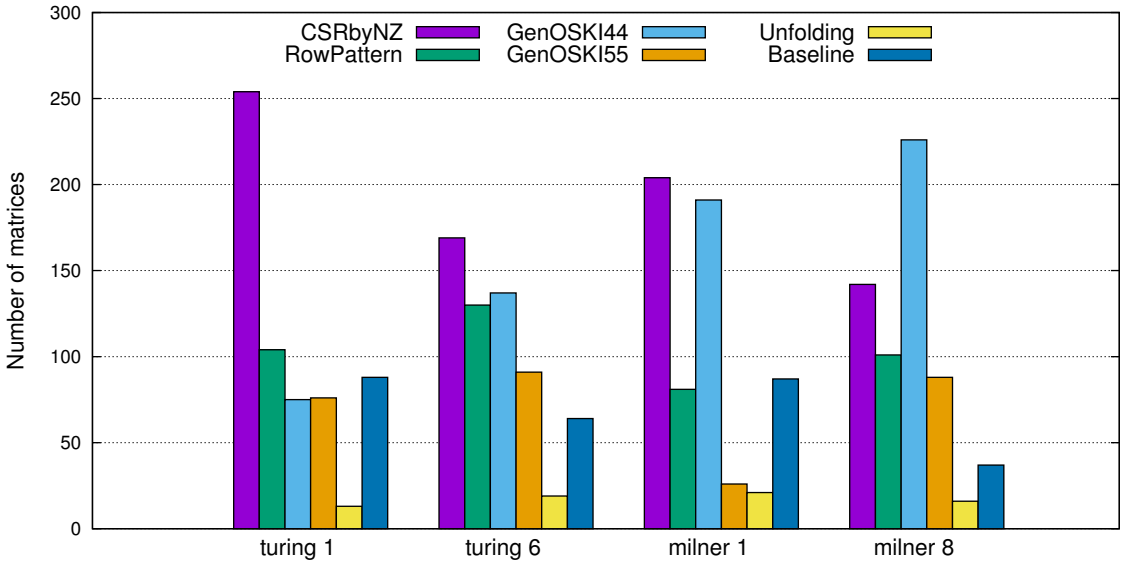


Figure 12: Number of times each method is the best (610 matrices in total).

5.2 Performance Results

In this section we discuss the performance results of the specialization methods. Figure 12 shows the distribution of best methods.

In Figure 12, it is seen that although the best method counts vary across machines and with different thread counts, *CSRbyNZ* and *GenOSKI* methods (both *GenOSKI44* and *GenOSKI55*) are likely to be the winning methods most of the time. *Unfolding* seems to fall behind in terms of the best method count, but it provides as much speedup as other specialization methods when it is a winning method. Except for *CSRbyNZ* and *Unfolding*, specialization methods seems to benefit from parallelization since their winning counts increase when moving from single-threaded to multi-threaded runs (i.e. from turing-1 to turing-6, and from milner-1 to milner-8). The percentage of the baseline implementation in Figure 12 is 10-14% for turing and 6-14% for milner. Hence, most of the time a specialization method is the best method.

Table 7 shows the average and maximum speedups w.r.t. the baseline performance when using the best method for each matrix.

Test Name	Avg. Speedup with the best method	Max. Speedup with the best method
turing-1	1.33	2.94
turing-6	1.45	4.28
milner-1	1.39	3.30
milner-8	1.83	5.69

Table 7: Average and maximum speedup w.r.t. the baseline performance when using the best method for each matrix.

In this work, our intention is to determine whether runtime specialization is feasible for SpMV. Therefore, we have not aggressively tuned our library to gain the best speedup we can achieve. We studied industry-strength compilers, icc and clang, and tried to mimic their optimizations. But, we believe, there is still room for improvement. Table 7 and Figure 12 show that runtime specialization can provide substantial speedup for SpMV. The maximum speedup that the baseline method can achieve over the best specialization method (not shown in the table) ranges between $2.00 - 2.62$.

5.3 Prediction Results

In this section we discuss the prediction results of the classifier. Figures 13 and 14 show the distribution of class labels when using the paired approach. In Figure 15 we show the prediction results for turing-1, turing-6, milner-1, and milner-8. For each, we show the number of *correct*, *semi-correct*, *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.

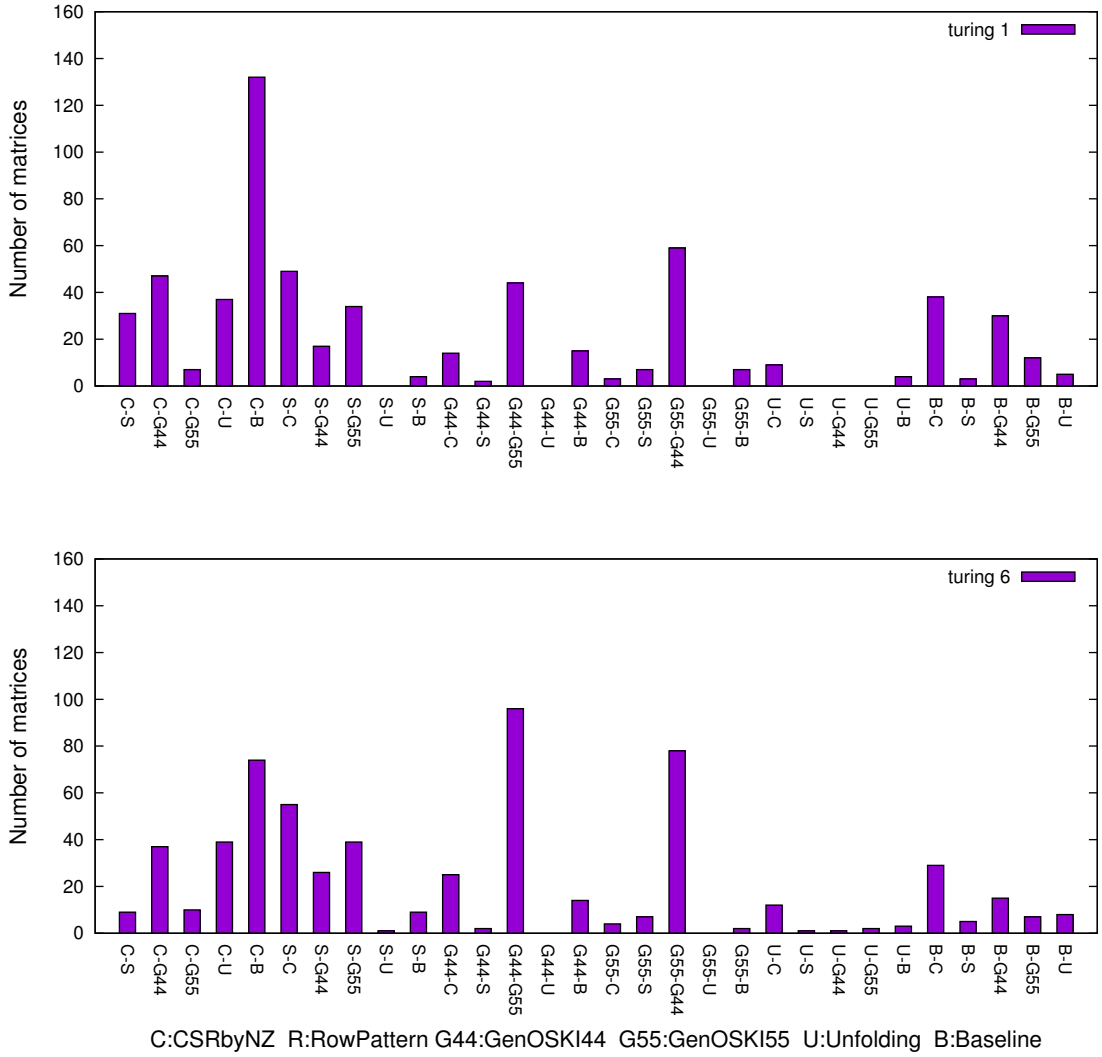


Figure 13: Class labels and corresponding counts for 610 matrices using the paired approach on turing.

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

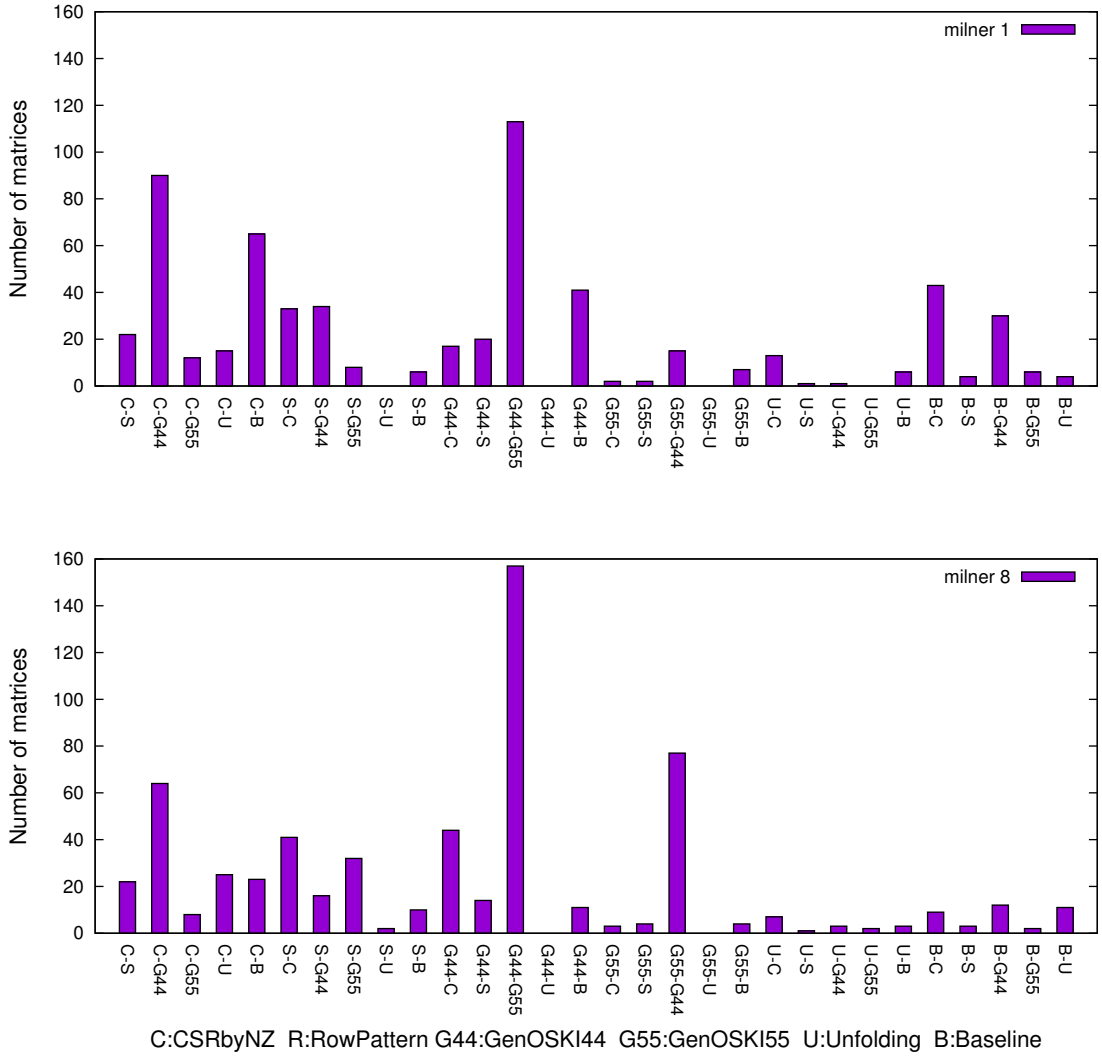


Figure 14: Class labels and corresponding counts for 610 matrices using the paired approach on milner.

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, we categorize this prediction as *correct*; if it is the same as the actual second best method, we categorize this prediction as *semi-correct*. Otherwise, the prediction is considered *incorrect*. In

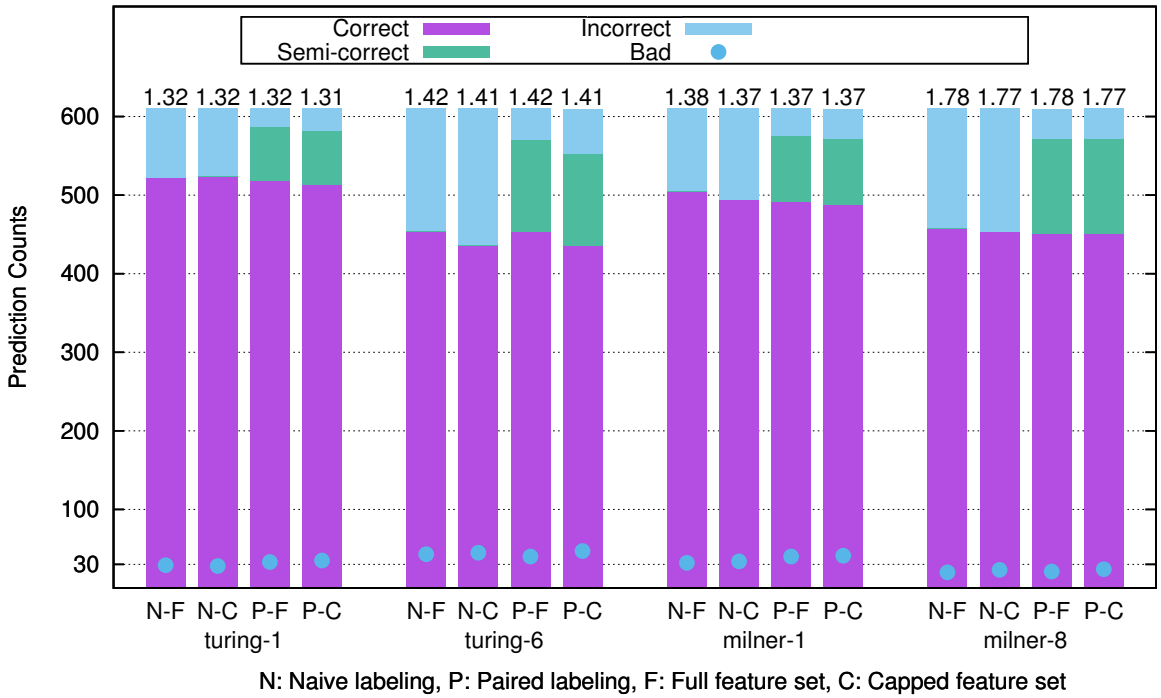


Figure 15: Prediction results.

both naive and paired labeling approach, an incorrect or semi-correct 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 15). The speedups are slightly better when using the naive approach or the full set. Recall from Table 7 that if always 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 trade-off.

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 4.3. 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 ($\sim$5 more) and bad predictions ($\sim$4 fewer). The achieved speedups did not change. However, the number of semi-correct and incorrect predictions were altered significantly. For instance, for turing-1, we obtained 69 semi-correct and 23 incorrect predictions when a threshold is *not* used, but 19 semi-correct and 67 incorrect when a threshold value of 1.02 is used. This is because some classes contain repeated method names (e.g. CSRbyNZ-CSRbyNZ) when a threshold is used. For those classes, there is no chance for a semi-correct prediction, a prediction is either correct or incorrect, according to our definition. We also examined the cases for which there is a large performance difference between the winning method and the runner-up: In our four experimental setups, there are 143-189 (24-30% out of 610 matrices) matrices where the winning method performs 1.20$\times$ the second best method. Among these, paired labeling approach gives semi-correct predictions for 3-14 matrices, and incorrect predictions for

2-9 matrices. For these reasons, we decided not to use the thresholding approach.

CHAPTER VI

LATENCY

SpMV specialization is likely to occur at runtime, unless the matrix (or at least its pattern) is available offline. If the matrix data is 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 chapter, 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 that specialization compensates its costs and starts to bring benefits? We show that, on the 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× or better speedup, we obtained average break-even points of 272 and 237 baseline SpMV operations on our testbed computers.

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, e.g.: 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), e.g.: for each block pattern in GenOSKI, a loop is generated.

- *Matrix conversion:* The matrix data is 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 Chapter 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.

We have parallelized *Matrix analysis*. The number of threads is set to number of threads set for threaded-runs. Although we also parallelized *Instruction emission* process, the results presented in this chapter are with single thread (denoted with turing-1 and milner-1).

6.1 Cost of Code Generation

Average costs of the code generation steps in terms of one baseline SpMV operation are given in Table 8. We provide two costs, "if best" and "overall", for each method. "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 short code is generated quicker. For instance, if there is a large number of row patterns in a matrix, both the analysis, instruction emission, and boiler-plate steps take 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 8 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

	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	

Table 8: Costs of code generation steps and feature extraction in terms of one baseline SpMV operation.

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 collec*ts 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

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 trade-off.

In Table 8 we provide values for only single-threaded execution on both turing and milner. The boiler-plate step is delegated to LLVM, and we are not sure if it can be parallelized, but it is possible for all the other steps and feature extraction to run concurrently by splitting the matrix and the analysis data into partitions.

6.2 Break-even Points

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 that specialization compensates its cost, and starts to bring advantage over the baseline implementation? Figure 16 shows the distribution of break-even points on turing and milner. 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 hundreds or a few thousands is common in practice. Considering this fact, the break-even points shown in Figure 16, 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. This, along with bad predictions, is shown in Table 9. For those matrices, no break-even point exists and no cost other than the feature extraction

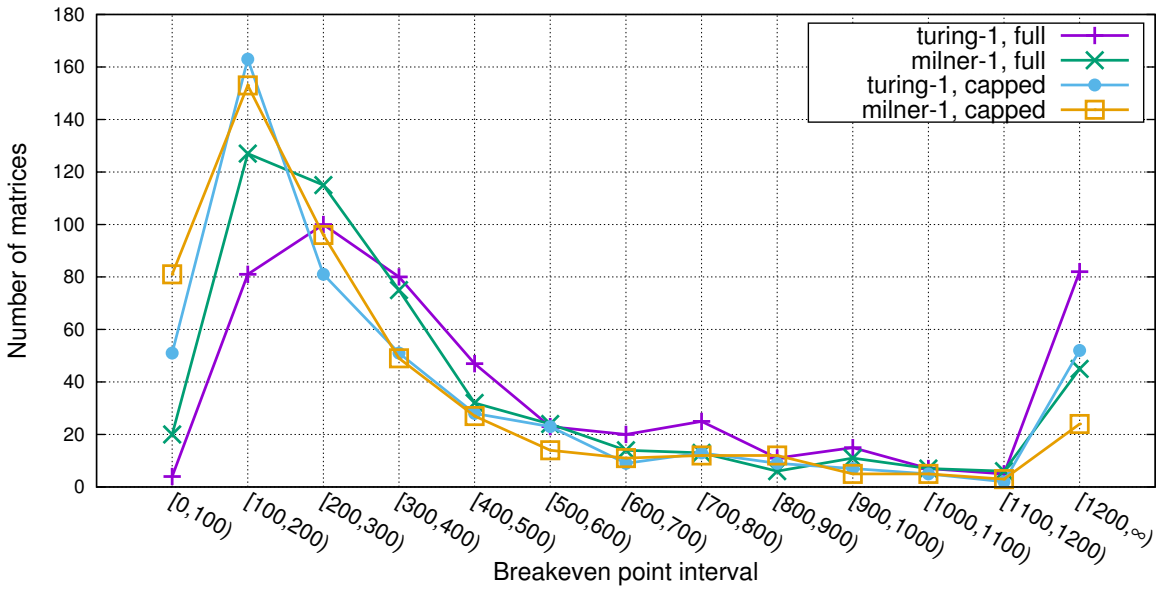


Figure 16: Distribution of break-even points of the predicted methods.

	Bad Predictions		Baseline Predicted	
	turing-1	milner-1	turing-1	milner-1
Full	33	40	77	75
Capped	35	41	81	77

Table 9: Count of bad predictions and baseline predictions for full and capped feature sets using single thread on both turing and milner.

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 [9]. They report these break-even points for matrices for which at least 1.1× speedup was observed (39 out of 53 matrices). Because we do analysis for 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× speedup (when the capped feature set is used and code generation cost is *included*) is 272 on turing (414 cases

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

Table 10: Count and break-even points of predictions that yield 1.1x or better speedup.

out of 610) and 237 on milner (432 cases out of 610), also shown in Table 10. 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.

CHAPTER VII

ADDITIONAL OPTIMIZATIONS

We have experimented with two additional optimizations on the generated code, vectorization and common subexpression elimination, to see their effect on the performance. In this chapter we discuss these two optimizations.

7.1 Vectorization

In our experiments with several compilers (icc, gcc and clang), we have seen that depending on the matrix's shape and the specialization method, the generated code can benefit from vectorization. Hence, we wanted to measure the impact of this optimization on the performance.

We experimented with SSE2 (Streaming SIMD Extensions) instructions that operate on vector registers (eight 128-bit registers known as XMM0 through XMM7). Since XMM registers are 128 bit wide, they can hold two double-precision floating point numbers. SSE instructions allow performing calculations with the two halves of XMM registers simultaneously.

We can store two consecutive column indices in an XMM register. We used MOVAPD and MOVUPD instructions to read two packed column indices from memory to an XMM register. These instructions allow loading data from aligned and unaligned memory locations. We used an ADDPD instruction to add column indices together which are stored in XMM registers. An example is provided in Figure 17.

HADDPD instruction (horizontal add) is similar to ADDPD instruction but add contents of the registers horizontally, adding low and high quadwords of an XMM register horizontally together. If we take the example in Figure 17, `haddpd %xmm0, %xmm1` will add 8.0 and 6.0 together and store 14.0 in XMM0's high quadword, and will add

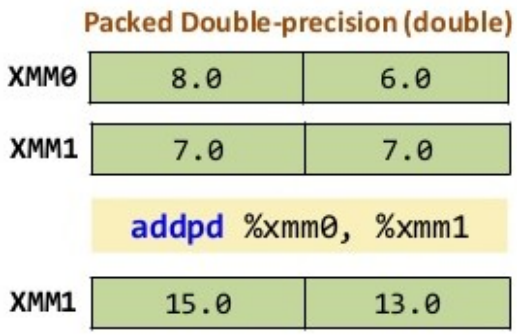


Figure 17: ADDPD instruction.

7.0 and 7.0 together and store 14.0 in XMM0's low quadword.

In *Unfolding*, all the array indices are explicit. Therefore, vectorization opportunities can be detected in a straightforward manner. Details of doing vectorization for *Unfolding* can be found in [90]. Other methods that vectorization is applicable are *RowPattern* and *GenOSKI* methods since both methods remove indirect indexing for column indices. However, we only did our experiments for *Unfolding*.

In our experiments, we have combined vectorization together with Common Subexpression Elimination (CSE), explained in the next section.

SpMV and vectorization-related work includes [9, 64, 91]. In these work, an external compiler is involved and code is generated at source level and vectorization is done using compiler intrinsics. On the contrary, we detected vectorization opportunities while generating code and emitted vector instructions appropriately.

7.2 Common Subexpression Elimination (CSE)

Common Subexpression Elimination (CSE) is one of the most well-known optimization passes of a compiler. It detects common subexpressions in arithmetic expressions to calculate these subexpressions only once and use multiple times. Multiplication instruction is known for taking many cycles to execute and thus, for keeping the ALU busy. Reducing the multiplication and addition instructions by CSE is expected to have a noticeable effect on the performance. How *Unfolding* can benefit from CSE

is discussed previously in Section 2.4. Here, we discuss the implementation of CSE as integrated into the code generation phase of *Unfolding*. Our implementation of CSE is customized to the context of *Unfolding* and hence differs from the traditional definition, such as the one found in [92]. We call this specialization method *UnfoldingWithCSE*.

We did not integrate *UnfoldingWithCSE* in our library because our focus in this dissertation is not to obtain the best SpMV speedup, but it is to show the feasibility of runtime specialization by using autotuning; the speedups obtained without CSE were sufficient for this purpose. We nevertheless present *UnfoldingWithCSE* here to give an impression of how much improvement could possibly be gained for *Unfolding*, at the expense of increasing code generation costs.

Below are the key properties of our CSE implementation:

- ***Application Level:*** We do not have a graph on which to perform the analysis. Instead, we analyze matrix rows to find column indices that are common among rows, and consider them as subexpressions that can be reused later.

- ***Type of common subexpressions:*** We restricted the common subexpressions to have consecutive column indices and to be multiplied with the same nonzero element of the matrix. This restriction is to make it easier to apply some arithmetic optimizations we discussed in Section 2.4, and for vectorization. This is demonstrated below in detail.

  ```
  w[0]=8*v[1]+5*v[3]+5*v[4]+5*v[5]+5*v[6]+8*v[10]
  w[1]=5*v[3]+5*v[4]+5*v[5]+9*v[11]
  ```

  In this example, a common subexpression is given. We will call a common subexpression as *CSEXP* in this context. Here, we consider two consecutive rows. We see that the expression `5 * v[3] + 5 * v[4] + 5 * v[5]` is common in both rows. The inverse of distribution of multiplication over addition

is an arithmetic optimization we perform in *Unfolding* (see Section 2.4). So the expression above becomes `5 * (v[3] + v[4] + v[5])`. To enable these optimizations *and* vectorization, we restricted CSEXPs to be multiplications of a constant matrix element with the sum of consecutive elements of the vector `v`; i.e. to situations that lead to the following expression format: c `* (v[`i`] + v[`$i+1$`] +...+ v[`$i+k$`])`. To do this, before CSE is applied, we group the row elements according to the distinct nonzero values of the row. At the end of the CSE analysis of two consecutive rows, a CSEXP object is created holding column indices and a value index for the nonzero value. The following subsection explains the CSEXP object.

- ***Scope of CSE analysis:*** Unlike the classical CSE algorithm, we do not analyze the whole search space and find every possible common subexpression. We rather focus on consecutive rows and search for common subexpressions locally within a sliding window of two matrix rows. This is decided based on the observation that common subexpressions usually appear on the rows close to each other. Banded matrices are an example to this. With this limitation, we also reduce the analysis cost imposed by CSE.

Since we apply arithmetic operations like inverse distribution of multiplication to take advantage of few distinct values, we vectorize only the accesses to the input vector `v`. For instance, an expression such as `5 * (v[4] + v[5] + v[8] + v[9])` is converted to the following pseudo assembly code:

```
mov v[4-5], %xmm0
mov v[8-9], %xmm1
addpd %xmm0, %xmm1
haddpd %xmm1
mul $5, %xmm1
```

CSEXP Object:

A CSEXP object is an object that four fields (i.e. it is a four-tuple) that we use to represent a common sub-expression in this format: c `*` `(v[`i`]` `+` `v[`$i+1$`]` `+`...`+` `v[`$i+k$`])`. Here, the nonzero constant c is a value loaded from the unique value pool; therefore, the format is in fact `vals[`j`]` `*` `(v[`i`]` `+` `v[`$i+1$`]` `+`...`+` `v[`$i+k$`])`. A CSEXP object has the following fields:

- Value index (i.e. j)
- Column indices (i.e. $i, i+1, \ldots, i+k$)
- Target register (i.e. `xmm0` through `xmm15`)
- Availability (i.e. true or false)

Here, availability denotes whether the summation part of the expression (i.e. `(v[`i`]` `+` `v[`$i+1$`]` `+`...`+` `v[`$i+k$`])`) is already computed. Target register denotes where the result of the summation will be stored. This representation allows us reuse a CSEXP in a bigger CSEXP and regardless of the multiplied nonzero constant. E.g. `5 * (v[3] + v[4] + v[5])` can be used to construct `9 * (v[3] + v[4] + v[5] + v[6] + v[7])`. All we need to do is to add `v[6] + v[7]` to `v[3] + v[4] + v[5]` and multiply it with `9`. If, in the CSEXP object, we stored the result of the multiplication `5 * (v[3] + v[4] + v[5])` instead of only the summation `v[3] + v[4] + v[5]`, this reusability would not have been possible.

UnfoldingWithCSE

We integrated CSE together with vectorization into *Unfolding*. We call this specialization method *UnfoldingWithCSE*. In *UnfoldingWithCSE*, instead of looking for common subexpressions in the entire search space, we limit the analysis to two consecutive rows in a sliding-window fashion. The motivation behind this choice is that

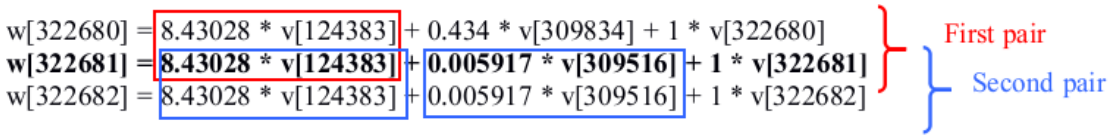


Figure 18: Pairwise CSEXP analysis for row 322681 of matrix webbase-1M.

CSEXPs are more likely to appear in rows close to each other, in particular, in banded matrices. Thus, each row of the matrix, except the first and the last, are analyzed twice: once with its predecessor and then with its successor. When generating code for a row, we favor the CSEXPs of whichever pair yields more coverage of elements. This way, we only need a one-pass over the matrix data to detect CSEXPs, and we substantially reduce the analysis cost when compared to a traditional CSE.

An example of the sliding window CSEXP analysis is given in Figure 18. Here, the values are from the webbase-1M matrix. Row 322681 is being analyzed once with its predecessor and once with its successor. In the first pair, a single common subexpression is found: $\langle 8.43028, v[124383] \rangle$. In the second pair, two common subexpressions exist: $\langle 8.43028, v[124383] \rangle$ and $\langle 0.005917, v[309516] \rangle$. CSEXP coverage of first the first pair is one since we have only one common column index, CSEXP coverage of the second one is two. When emitting code for row 322681, the analysis result of the second pair is favored because its CSEXPs cover more elements.

Evaluation

For the evaluation of *UnfoldingWithCSE*, we chose a small subset of our matrices that have very few distinct values and contain CSEXPs whose characteristics are provided in Table 11. For some matrices, although they have few distinct values, *UnfoldingWithCSE* did not find any CSEXPs. We omitted those cases here.

matrixName	N	NZ	Distinct vals	% Dist. vals	Avg NZ/row
cage12	130,228	2,032,536	350	0.01722	15.61
engine	143,571	2,424,822	1	0.00004	16.89
m133-b3	200,200	800,800	2	0.00025	4.00
soc-sign-Slashdot081106	77,357	516,575	2	0.00039	6.68
webbase-1M	1,000,005	3,105,536	565	0.01819	3.10

Table 11: Matrices selected for experimental evaluation of *UnfoldingWithCSE*.

Both m133-b3 and soc-sign-Slashdot081106 have 2 distinct values: 1 and -1. These two matrices are good candidates for the arithmetic optimizations considered in *Unfolding* since multiplication with 1 and -1 are transformed to addition and subtraction. engine has only one distinct value, 8.

In our experiments, we observed that for some matrices, putting a lower bound to the length of a CSEXP by using a threshold is beneficial. This decreases the number of CSEXPs created, but eliminates very short ones that are not likely to be useful. We give the threshold's affect on the number of CSEXPs created, on the number of column index pairs vectorized, and on the performance, in Table 12. To understand CSE and vectorization's affects on the performance better, one should find out more about common subexpressions of a matrix. For this, we also created a CSEXP histogram for each matrix in our set. In Tables 13, 14, and 15, we present the length and count of CSEXPs detected by *UnfoldingWithCSE*.

For engine, when the threshold is 3, CSEXPs with at least of length 4 are created. We see that the number of CSEXPs created are almost halved. Although the number of vectorized pairs are slightly increasing when the threshold goes from 0 to 3, at 3 it dropped down. Obviously, column indices became non-consecutive in this case. It is seen that the runtime dropped by 18% (i.e. the performance is improved) when the threshold is 3.

cage12				
threshold	0	1	2	3
num. of CSEXPs	28,937	0	0	0
num. of vectorized pairs	95,189	95,299	95,299	95,299
runtime	3,062	3,073	3,072	3,064
engine				
threshold	0	1	2	3
num. of CSEXPs	258,474	237,304	216,050	113,914
num. of vectorized pairs	564,896	573,492	585,482	524,779
runtime	2,469	2,435	2,437	2,026
m133-b3				
threshold	0	1	2	3
num. of CSEXPs	75,790	0	0	0
num. of vectorized pairs	0	0	0	0
runtime	6,998	7,421	7,412	7,423
soc-sign-Slashdot081106				
threshold	0	1	2	3
num. of CSEXPs	7,952	204	35	7
num. of vectorized pairs	30,210	30,852	31,043	31,086
runtime	3,417	3,443	3,477	3,509
webbase-1M				
threshold	0	1	2	3
num. of CSEXPs	142,549	9,450	1,716	879
num. of vectorized pairs	83,232	67,646	78,188	80,189
runtime	4,732	4,972	4,958	4,949

Table 12: Effect of threshold on number of CSEXPs, number of vectorized pairs and runtime.

However, there might be cases when increasing the threshold is not beneficial for performance. m133-b3 is an example of this. As seen in Table 12, m133-b3 does not benefit from vectorization at all. All we can do is make use of CSEXPs if there is any. When no threshold is applied, *UnfoldingWithCSE* created 75,790 CSEXPs. From its histogram, given in Table 14, we see that all of these CSEXPs are of length 1. Hence, when the threshold is set to a value higher than 1, we eliminate all CSEXPs resulting in a 6% decrease in performance. webbase-1M also behaves similar to m133-b3, with 1,415,278 CSEXPs of length 1 (see Table 15).

For cage12, the number of vectorized pairs and performance are almost not affected; however, the number of CSEXPs drop to zero when increasing the threshold values, since the length of the CSEXPS is 1 (Table 14). To our surprise, although

engine							
length	**count**	**length**	**count**	**length**	**count**	**length**	**count**
1	21,058	6	98,990	11	65	16	2
2	21,058	7	269	12	6,363	17	2
3	184,997	8	269	13	10	18	328
4	26,453	9	15,850	14	10	21	32
5	26,453	10	65	15	1,925	24	6
27	8					**Total**	404,213

Table 13: CSEXP histogram of engine.

CSEXP length	**soc-sign-Slashdot081106**	**m133-b3**	**as-caida**	**cage12**
1	15,953	127,158	3,308	29,056
2	241		7	
3	31			
4	8			
5	2			
Total	16,235	127,158	3,315	29,056

Table 14: CSEXP histogram of soc-sign-Slashdot081106, m133-b3, as-caida, and cage12.

cage12 has 28,927 CSEXPs, it did not benefit from CSE at all.

soc-sign-Slashdot081106 and webbase-1M are not affected significantly by the threshold in terms of performance, although there are drastic changes in their number of CSEXPs and vectorized pairs. Both soc-sign-Slashdot081106 and webbase-1M mostly have CSEXPs of length 1. However, setting the threshold to 1 affected their CSEXP and vectorized pair counts differently. This is because the sparsity regime of the matrix plays a very important role here. A matrix with many consecutive column indices benefits from vectorization. Also, distribution of CSEXPs in the matrix is an important factor on their reusability and the impact on the performance. We believe that analyzing a small part of the matrix beforehand (i.e. matrix sampling [7]) may reveal if CSE (and/or vectorization) will be beneficial.

In Table 16, we present performance comparison of *Unfolding*, *UnfoldingWithCSE* and icc. We omitted other matrices because for only engine and soc-sign-Slashdot081106 *Unfolding* was the winning method. To see CSE's and vectorization's effects, we compared *Unfolding* to *UnfoldingWithCSE*. And to see how good our implementation is,

webbase-1M					
length	**count**	**length**	**count**	**length**	**count**
1	1,415,278	13	60	29	3
2	30,675	14	15	31	1
3	6,082	15	19	32	2
4	2,179	16	46	34	9
5	1,195	17	71	39	1
6	623	18	32	40	1
7	519	20	10	42	1
8	179	21	37	43	1
9	328	23	33	44	12
10	136	24	20	45	43
11	82	25	91	52	1
12	48	26	34	302	3
				Total:	1,457,870

Table 15: CSEXP histogram of webbase-1M.

Matrix	*Unfolding*	*UnfoldingWithCSE*	icc
engine	4,960.15	2,026.27	1,874.86
soc-sign-Slashdot081106	842.16	679.99	832.76

Table 16: Runtime comparison of *Unfolding*, *UnfoldingWithCSE*, and icc.

we compared *Unfolding* to icc. The implementation that is used with icc is the one presented in [1]. We used `-O3` as the optimization level. We see that *UnfoldingWithCSE* improves the results (by 2× for engine), and becomes competitive with icc. However, icc still beats *UnfoldingWithCSE* for engine. Note that icc employs other optimizations such as register allocation (which we do manually) and instruction reordering (we do not do this optimization). For soc-sign-Slashdot081106, *UnfoldingWithCSE* improves the performance by 18.4%.

Also we provide the ratio of code generation performance of *UnfoldingWithCSE* and *Unfolding* for all matrices in Table 17. It is seen that CSE with vectorization increases code generation time significantly.

UnfoldingWithCSE is a specializer that we developed to understand the effect of CSE and vectorization. Our motivation was to see, in our context, to what extend

Matrix	*UnfoldingWithCSE / Unfolding*
cage12	0.32
engine	0.45
m133-b3	0.31
soc-sign-Slashdot081106	0.49
webbase-1M	0.31

Table 17: Code generation performance ratio of *UnfoldingWithCSE* to *Unfolding*.

CSE and vectorization can be integrated to a code generator. This increases the developer the effort and the complexity of the specialization method; however, delegating CSE optimization to a compiler is likely to increase the runtime costs well beyond the limits of practical benefits.

Previous work on code generation for algebraic expressions and optimization with CSE are [93, 94]. In these work, algebraic expressions are expressed as matrices and factorization is applied to find common subexpressions. We could not take the approach as an example to ourselves since we want to do CSE as quick as possible and compress the data transferred as much as possible. Work on vectorization can be found in [9, 64, 91]. In these work, code is generated at the source level and an external compiler is involved; vectorization is done using compiler intrinsics.

CHAPTER VIII

RELATED WORK

SpMV is a popular computational kernel that has been well-studied. This chapter covers not all but many research papers in this area categorized as autotuning and code generation. Under this categorization, both work aiming SpMV directly and more general examples of autotuning and code generation tools are presented. Finally, work focusing both on autotuning and code generation is given.

8.1 Autotuning

A variety of library generators use auto-tuning to generate sparse and dense linear algebra kernels, such as PHiPAC [95], ATLAS [61], and SPARSITY/OSKI [18, 60]. These libraries employs empirical search for tuning parameters. ATLAS and PHiPAC generate and tune kernels for dense matrices. Both SPARSITY and its successor OSKI use a model to estimate the performance of a given blocking factor based on architectural and matrix-specific parameters. Similar to ours, a hybrid offline/run-time empirical search based autotuning approach is taken. pOSKI [4] builds on prior work on the OSKI and targets both uniprocessor and multicore machines. It provides parallel functionality, and includes additional optimizations to OSKI. pOSKI supports several parallel programming models to create multiple threads on multicore architectures, and it also supports several partitioning schemes to split a matrix into submatrices. Other well-known uses of auto-tuning include FFTW [75], a library that produces Fast Fourier Transform code, and SPIRAL [76], which generates digital signal processing routines using a symbolic algebra system and genetic algorithms as a search method.

Previous autotuning approaches for SpMV focus mostly on choosing an optimal

storage format, because even the basic sparsity regime of a matrix can have profound effect on the performance [11]. To this end, there exist work using decision trees [81], dynamic-programming [96], reinforcement learning [79], heuristic-based autotuning [83], and model-driven approaches [82, 26, 97].

Li et al. [81], introduce SMAT: an auto-tuning system providing both application and architecture aware SpMV kernels. User provides the input matrix in CSR format and SMAT determines the optimal format (among CSR, COO, DIA and ELL) and implementation on a given architecture using machine-learning. SMAT uses a two-staged approach being off-line and online for autotuning. Off-line stage consists of optimal format and implementation search using machine-learning techniques. If this stage fails to meet the requirements for a successful guess, online stage is triggered. In this stage, available candidates are benchmarked and the one with the highest performance is outputted. They report prediction accuracy of 82% on both Intel and AMD architectures.

In [96], Guo et al. present a profile-based performance modeling and optimization analysis tool to predict and optimize performance of SpMV on GPUs. A dynamic-programming based autotuning algorithm is proposed to report on optimal storage strategy, storage format(s) and execution time. Formats considered are CSR, ELL, COO and HYB. First, execution times are collected from benchmark matrices. Then, along with properties, the execution times are used to instantiate models used for estimating performance of a SpMV kernel. In the model, partitioning the matrix and using different storage format for each is also considered.

Armstrong et al. [79], use reinforcement learning to automatically choose between COO, CSR, BCSR using number of rows, number of columns, number of nonzeros, standard deviation of of nonzeros per row and number of neighbors as matrix features. The learning agent is characterized by two parameters: an exploration rate and a parameter that determines how the state space is partitioned.

Sufah et al. [83], describe an autotuning framework for both selecting the optimal storage format (among ELLPACK, BELLPACK, DIA, HYB and their own format BTJAD) and CUDA parameters for SpMV kernels on GPUs. The autotuning framework uses heuristics to guide the search and it consists of two stages: Reduce candidate storage formats based on matrix sparsity characteristics and select kernel parameters based on matrix characteristics and/or specifications of the target GPU. Pruning kernel parameters is partly based on architecture specifications and matrix specific parameters like block size.

In [82], Neelima et al. propose a model-driven approach to choose an optimal sparse matrix format for SpMV. Authors suggest that for predicting a suitable storage format for SpMV on GPU, besides matrix features, one should also consider communication overheads and transforming time caused by the chosen format. Matrix features are used for categorizing the matrix into different sparseness levels and determining the non-zero elements distribution. Then, data sizes for each format (CSR, COO, ELLPACK and HYB) are computed. Using data size and PCI-E bandwidth capability CPU to GPU communication time is also calculated. Based on this observation optimal model is chosen.

In [26] Choi et al. introduce new storage format called BELLPACK and a performance model based on model-driven approach to guide tuning. It is based on offline benchmarking and a model that is instantiated at run-time. Kernel specific parameters are determined by offline benchmarking. This information is fed to the model and performance is calculated.

In [97] Guo et al. present a performance-model driven approach for partitioning sparse matrix into appropriate storage formats (COO, CSR, ELL, HYB) and an autotuning framework to choose appropriate CUDA parameters. Auto-tuner benchmarks predefined matrices to instantiate parameters of the performance model at offline stage. Matrices are sorted based in number of non-zeros per row. At runtime, using

the performance model, the incoming matrix is optimally partitioned by means of load balance and each partition is transformed into specific format. Meanwhile, the system adjusts CUDA parameters automatically by invoking all possible combinations of parameters on the specific architecture and measure their performance in the first iteration.

Su et al. [34] introduce a new format called Cocktail Format which partitions a matrix into submatrices and stores them in best storage format falling into categories flat, blocked and diagonal. The storage formats considered include DIA, BDIA, ELL, CSR, COO, BELL and BCSR. The framework analyzes a given matrix at runtime and predicts the best representations for different platforms. It benchmarks in off-line phase using different sparse matrix settings such as matrix dimension and number of non-zero elements and estimates performance. It solves Cocktail Matrix Partitioning problem at runtime using greedy approaches using features collected for different matrix formats.

In [78], Zein et al. study for SpMV, how to determine the best CUDA implementation for a given format. The goal is to come up with a blackbox implementation that can determine this best CUDA implementation from some characteristics of the sparse matrix. Using a code generator, all possible implementations are produced and their performance is measured. To select among best performing implementations, first a best implementation set (BPS) is selected among all implementations using greedy approach. It is constructed incrementally by adding the one additional implementation that gives rise to the biggest performance enhancement. Then, given a BPS, a decision tree with attributes number of rows, number of columns and average number of non-zeros per row is used to select the best implementation.

In [98], Yang et al. optimize graph-based data mining algorithms such as PageRank, HITS, and RandomWalk on GPUs whose key computation is spMV. The work

focuses on large graphs - typically with power-law characteristics. The authors propose a composite storage format which combines CSR and ELL and use tiling. Based on empirical observations, problems causing memory traffic are addressed with several optimizations. The framework can be used on both single- and multi-GPU systems. Balancing the workload are determined with a performance model which relies on executing model of CUDA kernels. Also automatically tunes parameters for tiling using a greedy heuristic.

A new storage format for sparse matrices named blocked compressed common coordinate (BCCOO) is introduced in [24] to address bandwidth problem. Yan et al. propose a matrix-based segmented scan/sum approach to address load imbalance problem. They also propose an auto-tuning framework to choose optimization parameters based on characteristics of the input sparse matrices and target hardware platforms. Parameters include performing transpose online or offline, suitable block size, number of nonzero blocks to be processed, number of threads in a workgroup, and size of shared memory. The search space is pruned using heuristics. Also compiled kernels are cached for future use. OpenCL code is generated as the result of auto-tuning.

Li et al. present performance analysis based on probability mass function (PMF) for SpMV on GPUs [99]. PMF is used to analyze distribution pattern of non-zeros in the sparse matrix giving its compression efficiency. Combined with hardware parameters of the target platform, PMF is used in performance estimation of the SpMV kernel with different storage formats among COO, CSR, ELL, HYB and predicting the most appropriate format for the input matrix. Since the analysis is based on a mathematical model, no benchmarking is needed.

To the best of our knowledge, our work 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

Support Vector Machine (SVM) based approach for autotuning. SVM is used in many autotuning systems including the Nitro framework [66] and others [22, 63, 64, 65].

Most of the other autotuning work have smaller matrix sets than ours, e.g. ∼14-150 [100, 66, 82, 96]. There also are studies with bigger matrix sets, e.g. ∼2000 in [81], 1000 (synthetic) in [79].

In [8], Williams et al. discuss several optimizations that are effective on different multicore platforms to improve SpMV performance. Optimizations discussed are divided into three categories: Low-level code optimizations, data structure optimizations and parallelization optimizations. Examples of optimizations provided are software pipelining, branch elimination, pointer arithmetic, prefetching, blocking (register, cache, TLB) and threading. Authors compare their work to OSKI [60]. The framework uses a perl-based code generator to generate optimized spMV kernels for the low-level optimizations considered. Their auto-tuning framework uses a heuristic to minimize memory traffic and to select the appropriate parameters for tuning such as block size. Additionally, the best prefetch distance is exhaustively searched on each of their testbeds.

Vuduc et al. present ways of developing search-based systems for automatic tuning in [22]. It proposes solutions to early-stopping problem and run-time implementation selection. They develop a heuristic to stop an exhaustive compile-time search early if a nearly-optimal implementation is found. The algorithm is based on user defined search tolerance parameters. The method performs a partial search while still providing an estimate on the performance of the best implementation. In addition, they show how to construct run-time decision rules based on runtime inputs to select best implementation. The problem is formulated as statistical classification task in which a set of decision rules are automatically constructed at runtime to select the best implementation. Three models are used: Linear regression, separating hyperplanes and support vector machines (SVM) and report on their misclassification rates.

In [101], Demmel et al. introduce two systems:ATLAS for dense and BeBOP for sparse linear algebra kernels. Both use heuristic search strategies to explore architecture parameter space. ATLAS generates different kernels, applies optimizations such as software pipelining, register blocking, loop unrolling and then measures the execution times and compares them. For the sparse case, performance also depends on input matrix's non-zero structure. Hence, most optimizations must be deferred to runtime since matrix might not be known till runtime. The auto-tuning framework is a combination of off-line benchmarking and heuristic performance modeling at run-time. Benchmarks are run on the target machine and at run-time performance relative structural properties of the matrix are estimated and these are combined to predict the implementation for the best performance. Various optimizations are considered including register blocking, cache blocking, variable block splitting and exploiting diagonal structure. For determining the best implementation for a matrix, a probability density function based classifier is used. For a new matrix, given its feature vector method is found which maximizes the prob. density function. For feature extraction and characterization of algorithms multi-variate Bayesian decision rule is used.

POET (Parameterized Optimization for Empirical Tuning) [102] is an embedded scripting language that allows for parameterizing complex code transformations so that they can be empirically tuned. It aims at improving generality, flexibility and efficiency of existing empirical tuning systems. Yi et al. show loop optimizations: interchange, blocking and unrolling. POET can be embedded in code written C, C++ or Fortran by treating input code fragments as parameterized strings without the need to interpret the underlying language. The paper presents an example for tuning optimizations for matrix multiplication for both sparse and dense cases.

A model-based approach for memory hierarchies to optimally block matrices is proposed in [103]. The approach aims to amortize the cost of moving data between

different levels of the memory. Hence, the framework suggests a family of algorithms based on input shape which is locally optimal to each level of memory hierarchy. It derives a cost model for the memory hierarchy starting from registers and various blocking strategies of a matrix.

In [104], Lee et al. present optimizations for SpMV and SpMM when the matrix is symmetric and an empirical modeling and search based auto-tuning approach for selecting tuning parameters. Parameters to be tuned for SpMM are block size r,c and vector width v. After off-line benchmarking per machine, performances are measured for SpMM for a dense matrix stored in sparse format for all (r,c,v). Then, at run-time true fill ratio of the matrices measured. Last, (r,c,v) that maximizes estimate function of register blocking performance is chosen.

In [105] Guo et al. propose an auto-tuning framework for selecting optimal CUDA parameters for CUDA CSR kernel. Parameters such as number of threads and number of block size are chosen based on GPU architecture. Warp size is chosen based on the input matrix. After combining all these parameters, CSR kernel is invoked with different combinations. After the first iteration, the best combination is determined and rest of the iterations are done using this combination.

In their study, Reguly et al. [106], focus on efficient implementation of SpMV on cache-based GPU architectures. They introduce a model to select near-optimal parameters such as number of threads per block with special attention on caching mechanisms. Based on this, a dynamic run-time auto-tuning system to improve the performance is introduced. And a fixed rule in which preset values for parameters are chosen using exhaustive search is proposed. Fixed rule sometimes fails to deliver near-optimal performance since characteristics are unknown until runtime. Hence, an empirical algorithm based on exhaustive search is proposed.

Following work consist of autotuning for kernels other than SpMV or for general compiler optimizations. First couple of work are examples of using machine learning

techniques for optimization.

In [107], Li et al. study machine learning techniques to extend empirical search for sorting algorithms. The library generator produces implementations of composite sorting algorithms that are specialized for input characteristics and architecture of the target machine. It uses genetic algorithms and a rule-based classifier to search for the optimal sorting algorithm.

In [108], Monsifrot et al. address automatic generation of optimization heuristics on a target processor by machine learning. Authors target compiler optimizations in general but focused on loop unrolling in this study. Unrolling decision rules are represented as oblique decision trees. The classifier is binary since it decides whether to apply loop unrolling or not. Loop abstraction features used in the classifier are memory access count, arithmetic operation count, size of loop body, control statement in loop and number of iterations. These include both architecture-specific and -independent features whereas we used only matrix related features in our work.

In [63], Stephenson et al., discuss how machine learning techniques can be used in heuristic tuning in general. The authors focus on loop unrolling. Near Neighbor (NN) Classification and Support Vector Machines (SVM) are used to predict unroll factors. Open Research Compiler [109] is used as testbed. It uses two loop unrolling heuristics by toggling software pipelining. The feature set for supervised learning contains loop characteristics such as trip count of the loop and number of operations in the loop body. Among 38 such features a subset that improves the classification accuracy is chosen using Mutual Information Score (MIS) and Greedy Feature Selection (GFS). The subset of features is the union of top 5 features selected using MIS and GFS. Multi-class classifier has 8 classes corresponding to 8 different unrolling factors. The supervised learning is trained offline. For accuracy leave-one-out cross validation is used. In our work, we used features unique to our specialization methods as well as common features such as number of rows and number of nonzeros. We have as many

classes as the specialization methods and the baseline method. Like Stephenson et al., we use SVM and train our classifier offline and use LOOCV for accuracy measurements.

Cavazos et al., in [110], propose to use performance counters to determine good compiler optimization settings by using machine learning. The model examines performance counters of a new program and by using prior knowledge from previous programs determines a set of optimizations that are most likely to result in a speed up. Performance counters include cache hit, cache miss, branch prediction statistics. The learning model is based on logistic regression and it predicts best transformation sequence for the input performance counter values. Study cases include various benchmarks written in C, C++ and Fortran.

Another approach for auto-tuning optimization parameters is introduced in [111]. Ganapathi et al. propose to use static machine learning techniques: Kernel Canonical Correlation Analysis (KCCA) to search the space of tunable optimization parameters such as thread count, domain decomposition, software prefetching, padding and inner loop. KCCA finds multivariate correlations between optimization parameters and performance metrics and uses them to optimize performance and energy efficiency. Authors show their results on stencil code optimization.

In [64], Stock et al. address the problem of selecting the most effective combination of transformations for automatic vectorization on today's compilers using machine learning (ML) techniques. 6 different ML models from Weka are used: IBk, K*, M5P, SVM and LR with features extracted from generated assembly code. Models are trained offline and used at compile-time to choose among generated variants. Training of ML models are done on tensor contraction kernels and stencil computations. The code generator generates vector intrinsics after following optimizations: loop permutation, unroll-and-jam and choosing loop to be optimized. ML models are based on predicting the performance of vectorized codes without running them.

Assembly features considered are: vector operation count, arithmetic intensity, sufficient distance, sufficient distance ratio, total operations and critical path. In order to benefit from the ability of different models to predict best transformations for different benchmarks, a second-order model combining the predictions of the two best individual models is used. Also, using linear regression a weighted rank model is developed to output the variant that ranked first.

In [65] Trouve et al. focus on using machine learning techniques to decide whether it is beneficial or not to apply basic block vectorization to obtain speedup. Vectorization sometimes causes a slowdown. Authors use SVM and demonstrate that it significantly improves the quality of the code produced by Intel Compiler. Proposed approach relies on pattern matching and it focuses on loop unrolling. They conducted two experiments differing only at inputs: Already transformed program to find out if it were beneficial to do so or not and untransformed program along with an unrolling factor. Feature extraction is done at IR and AST level, with 12 static and hardware-independent features. This data is fed into SVM where training and test sets are 80% to 20% and quality is measured by LOOCV and plotting the learning curve.

Cavazos et al. develop a method-specific approach that automatically select the best optimizations on a per method basis within a dynamic compiler in [112]. The approach uses logistic regression to derive a model that determines which optimizations to apply on the features of a method. The technique is implemented in Jikes RVM Java JIT Compiler. Training data is generated by randomly applying different optimizations to methods and timing the program performances. 26 features are extracted denoting the optimizations applied.

In [113], Kamil et al. present a stencil auto-tuning framework that converts Fortran 95 stencil expression to tuned parallel Fortran, C,or CUDA. Optimizations such as loop unrolling and cache blocking are applied as transformations on the AST built

from the problem specification. To generate code for different platforms several back-end code generators are used. As part of the auto-tuning framework, strategy engines are used to enumerate optimizations with different parameters. These are run, timed and best is reported.

An optimization and auto-tuning framework for stencil computations is introduced in [114] by Datta et al. . Proposed approach is both for multicore machines and GPUs. Multi-threaded C code variants for different optimizations are generated using a perl based code generator. Then via benchmarking, parameter space is searched using several search heuristics chosen intuitively based on their performance experiments. Hence, auto-tuning framework outputs both peak performance and optimal parameters. Examples of parameters thread block size, core block size, register block size and DMA size. And optimizations include hierarchical blocking, unrolling, reordering and prefetching.

Autotuning unrolling factors has been studied widely using machine learning techniques [108, 63, 65]. Examples using other techniques are [115, 116].

In [115], a semi-automatic compile-time approach for identifying optimal unrolling loop factors for CUDA programs is proposed by Murthy et al. . To identify optimal unrolling loop factors, the framework analyzes compiled CUDA codes using Orio [117] and PTX analyzer of nVidia and estimate the performance of various unroll configurations.

In [116], Kisuki et al. address selecting tile sizes and unroll factor problem simultaneously by means of iterative compilation. The framework applies a sequence of transformations and decides the next transformation by a search algorithm such as genetic algorithm, simulated annealing or windows search. Code variants with different transformations are generated and benchmarked. After a number of iterations, the search algorithm outputs the transformation with the shortest timing.

Following work are other general frameworks for autotuning.

In [118], a specializer for the matrix powers kernel built by SEJITS[119] is introduced. Optimizations applied by the specializer are thread and cache blocking, tiling, symmetric representation and index array compression. First, the auto-tuner tries out possible optimization parameters and then optimized code is translated from Python to C using SEJITS. The auto-tuner benchmarks code variants for the best implementation and reports back. SEJITS uses runtime code generation as we do but it generates the same code with different parameters and benchmarks them to find the best implementation. On the other hand, we have multiple specializers and predict the best one using a multi-class classifier and generate code only for the predicted. Another difference is that they generate code at source-level while we do it in assembly-level.

In [117], Hartono et al. introduce Orio: an extensible annotation-based empirical tuning system which supports both architecture-specific and -independent optimizations. Orio generates many tuned versions of the same operation from annotated C code and selects the one with the best performance. Code transformations range from low-level loop optimizations to composed linear algebra operations such as memory alignment, loop unroll/jamming, loop tiling, register tiling, multi-core parallelization (using OpenMP). Orio also provides various heuristics (random search, Nelder-Mead simplex method and simulated annealing) to prune the search space to reduce the auto-tuning cost.

Jordan et al. introduce a multi-objective auto-tuning framework comprising compiler and runtime components in [120]. Framework focuses on individual code regions, computes a set of optimal solutions by using a multi-objective optimizer resulting in a multi-versioned executable. Hence, runtime system can choose among different versions dynamically adjusting to changing circumstances. Tunable parameters include tile size, unrolling factors and number of threads. The framework is implemented based on Insieme Compiler and Runtime Infrastructure [121]. Jordan et al. work on

loop tiling as a study case for their framework.

Active Harmony, an automated runtime tuning system allowing runtime switching of algorithms and library and application parameter tuning is proposed in [122]. Runtime switch of algorithms is based on a performance monitoring system. Tuning algorithm is a parallel algorithm based on simplex method.

A parameter prioritizing tool for Active Harmony[122] to help focus on performance critical parameters is described by Chung et al. in [123]. Each parameter is specified with minimum, maximum, default values and distance between two neighbor values. Using these, the tool tests the sensitivity of each parameter to determine the impact of change of the parameter on performance. Also historical data is used to speed up tuning.

Recently, a two-level approach to autotuning was shown effective to address the complexities of mapping features to algorithmic configurations [124]. We leave it a future work to see whether this approach improves the prediction accuracy for our experiments.

8.2 Code Generation

There exist several work that employ code generation for SpMV. Some of these work apply compile-time and some apply runtime specialization.

Willcock and Lumsdaine [33] generate matrix-specific compression/decompression and multiplication functions. The authors propose two compressed storage formats and their multiplication algorithms: DCSR and RPCSR. For DCSR, highly tuned decompression and multiplication routines are generated at different levels for different processors. Hence, generated code is aggressively tuned while compromising portability. For RPCSR, matrix-specific dynamic code which is again specific to the processor is generated at runtime in assembly language. Kourtis et al. [7] 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. Similar to our capped feature extraction approach to reduce matrix analysis cost, Kourtis et al. employ matrix sampling and show that it reduces costs by allowing minor loss speedups.

Sun et al. [125] introduce a runtime code generator for OpenCL that produces code variants for diagonal patterns for their CRSD format. Belgin et al. [9] propose a new format PBR which identifies recurring block structures that share the same pattern of non-zeros within a matrix. (The *GenOSKI* method we use is a variant of PBR.) A runtime code generator generates optimized custom kernel for each pattern. They generate code at the source-level and invoke an external compiler. They also have a code cache that can be used to dynamically link object files for existing, already-compiled code. They show that priming this cache with common block pattern code reduces runtime generation costs. Mateev et al. [126] 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 [100] by Grewe et al. 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 [127] is a compiler that generates native code at runtime from specifications given in a subset of ML. It derives a code generator from source code that contains expressions labeled with *late* and *early* annotations. Carette and Kiselyov [58] show how to eliminate abstraction overheads from generic programs using multi-stage programming on Gaussian elimination. Rompf et al. [57] propose to 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 [128].

We developed our code generator manually. It may be possible to derive it systematically from source code using a code generation/staging approach as in Fabius [127], LMS [57], or Tempo [129], but we have not tried this yet. It is unclear whether we can do code emission rapidly *and* produce high quality code using one of these approaches. As a trade-off, we comprise portability of our compiler.

Earlier examples of using code generation to optimize linear algebra operations include [130] and [131]. They generate machine code based on the matrix structure. Giorgi and Vialla [132] generate SpMV kernels based on characteristics of the input matrix. Venkat et al. [12] address indirect loop indexing and irregular data accesses in SpMV kernels and introduce new compiler transformations and automatically generated runtime inspectors. Our *RowPattern* and *GenOSKI* methods also eliminate indirect indexing. Neither of these papers do runtime generation. Belter et al. introduce a domain-specific compiler (BTO) in [133] to compile linear algebra kernels automatically by optimally combining several BLAS routines. To reduce memory traffic, BTO fuses loops of successive BLAS routines. It takes a matrix and vector arithmetic in annotated MATLAB and produces a kernel in C++. On the contrary to our compiler, BTO generates code at source level and applies some optimizations and passes the AST to a compiler graph to apply other low-level optimizations. An analytical model of the memory (cache and TLB) predicting the amount of data access for each instruction is used to differentiate between optimization choices and performance prediction.

There are several other examples of code generation frameworks for either general purpose or for other scientific computational kernels such as stencil computations or tensor contraction.

To bridge the performance gap between productivity-level languages (PLLs) such

as Python, MATLAB and efficiency-level languages (ELLs) like CUDA, Cilk and C with OpenMP, Catanzaro et al. propose use of just-in-time specialization PLLs in [119]. PLLs emphasize programmer productivity over hardware efficiency and ELLs lack the abstractions provided by DSLs. Code is dynamically generated in ELL within the context of PLL interpreter. Only those parts that will provide high performance improvement are generated in ELL compensating for runtime overhead. The JIT machinery is embedded in the PLL itself making it easy to extend.

Holewinski et al. present a code generation scheme for stencil computations on GPUs to decrease global memory bandwidth requirements in [134]. Several compiler algorithms are developed for automatic generation of efficient, time-tiled stencil codes. Input to the code generation scheme is a sequence of stencil operations described in their stencil DSL and it outputs overlapped-tiled GPU code and a host driver function written in C/C++.

In [91], Stock et al. describe a model-driven compile-time code generator that transforms tensor contraction expressions into highly optimized short-vector SIMD code. Since nested loops of tensor kernels can be fully permuted, a performance model to estimate relative number of execution cycles for different loop permutations is proposed. With the best loop permutation predicted by the model, unrolled loop C code with SSE intrinsics is generated. The code synthesizer doesn't generate assembly code directly to focus on vectorization and leave register allocation to the compiler.

Kamil et al. introduce Asp (SEJITS for Python) in [135], a framework to bridge the gap between productivity and performance. It embeds DSLs into Python for popular computational kernels such as matrix algebra and stencils providing a domain-specific but language-independent AST along with an optimization strategy. And it results in efficiency-level specialized code. SEJITS is a typical example to runtime code specialization.

8.3 Autotuning and Code Generation

Like our library, some research focus on both code generation and autotuning at the same time. Our library generates specialized code and uses autotuning to decide on the best specialization method.

In [136] Shin et al. discuss that auto-tuning frameworks like ATLAS and GOTO perform well for large matrices achieving 70% of peak performance but for small matrices achieves only 25%. For improving small matrix performances, optimizations should focus on loop overhead, managing registers and exploiting ILP. Hence aggressive loop transformations like loop permutation and unroll-and-jam are needed. The paper presents code specialization done using CHiLL[137] combined with an auto-tuning framework that uses heuristics to narrow down the space of different implementations. As a result of benchmarking, the system reports a library of implementations for a particular problem, domain and size. The framework is used to speedup Nek5000, a spectral-element code in [138]. On the contrary, our library focuses on large matrices, we use our own purpose-built code generator and our autotuner outputs a generative or non-generative method to generate specialized code for the given matrix.

In [139], PATUS a code generation for both CPUs and GPUs and auto-tuning framework for stencil computations is introduced. It takes three DSLs specifying the problem, optimizations like parallelization, explicit SIMDization and loop nest unrolling and hardware specifications. Then auto-tuner searches for the optimal parameters by running benchmarks and generating the kernel again and again based on a search method.

In [140], Hall et al. provide code transformation recipes for code generation in the form of specification of parametrized variants. Hence, it provides a common API for a compiler transformation (unroll-jam, tile, permute, split, fuse...etc) framework. Code generation is done using CHiLL[137] and POET[102] for OpenMP and CUDA

code. It is part the an auto-tuning framework which does benchmarking and selects an implementation that meets a set of criteria the best.

PetaBricks: A new implicitly parallel language and compiler is presented in [141]. The motivation is to have multiple implementations of an algorithm and multiple algorithms to solve a problem. It consists of a source-to-source compiler that translates from PetaBricks language to C++, an auto-tuning system that is based on genetic algorithm. The compiler performs static analysis and encodes algorithmic choices and tunable parameters in the output code. An algorithmic choice is a first class construct of the language. Choices include automatic parallelization techniques, data distribution, algorithmic parameters, transformations and blocking. The autotuner builds a multi-level algorithm in which each level consists of a range of input, corresponding algorithm and a set of parameters. Either this is run or fed back to the compiler.

Han et al. present Pattern-driven Stencil Compiler-based tool (PADS) in [142] which is a tool to reuse and tune stencil calculation kernels for different GPU platforms. It consists of OpenMP-to-CUDA translator, an optimized stencil template generator, a code generator with template library and a tuning system. C++ is used to rewrite stencil kernel codes incorporating domain-specific knowledge. Code generator generates CUDA code for different stencil patterns and parameters. Both platform -specific and -independent parameters such as blocking factors, grid thread size, loop unrolling are tuned by the tuning system. And template library is responsible for recording optimized template codes.

Tiwari et al. introduce a runtime compilation and tuning framework for parallel programs in [143]. Previous work on Active Harmony [122] is extended for tunable parameters that require code generation using CHiLL [137]. An online auto-tuner that can tune multiple code-sections simultaneously is proposed. The code generator, based on various parameters generates and compiles code on the fly. All generated

code-variants are sent to a parallel machine for auto-tuning. Auto-tuning is carried out in parallel. Parallel Rank Order (PRO) proposed by Tabatabaee [144] is used together with a penalization method for boundary constraints. Optimizations considered include loop unrolling, loop fusion, loop split and data-copy operations.

CHAPTER IX

CONCLUSIONS

In this dissertation we have shown that it is possible to use runtime specialization to form efficient SpMV in the context of iterative methods or when the same matrix is multiplied by several 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 also experimented with vectorization and common subexpression elimination (CSE): compiler optimizations that are observed to be beneficial for SpMV. We have shown that vectorization can be integrated into the code generation library by implementing emitting functions necessary and altering the specializers accordingly. Vectorization is available only *Unfolding*, *RowPattern* and *GenOSKI* methods. We implemented it only for *Unfolding*. We have showed that CSE can improve the SpMV code's performance significantly for some matrices, however, at the expense of substantial analysis cost. Hence, we did not include these optimizations into our code generation library.

We experimented with 5 specialization methods and also Intel's MKL. We evaluated two class labeling approaches and used SVM machine-learning technique to predict the best method to eliminate the need to produce many code variants. Our experimental results using 610 matrices and running on two different machines 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 also experimented with a capped feature extraction approach that reduces 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.

Lastly, let us give a brief discussion about what is on our schedule next: We would like to evaluate the performance results and better understand the bottlenecks and try to solve them. We also want to report on parallel code generation. At a larger scale, we aim to revisit vectorization, and add it to methods where applicable. Also, we will also consider kernels other than SpMV and lastly, we hope to port our code generation library and framework to GPUs.

References

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

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

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

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

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

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

[7] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: An extended compression format for spmv on shared memory systems," *SIGPLAN Not.*, vol. 46, pp. 247–256, Feb. 2011.

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrixvector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178 – 194, 2009.

[9] M. Belgin, G. Back, and C. J. Ribbens, "A library for pattern-based sparse matrix vector multiply," *Int. J. of Parallel Programming*, vol. 39, no. 1, pp. 62–87, 2011.

[10] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix-vector product computations using unroll and jam," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 225–236, May 2004.

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

[12] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *Programming Language Design and Implementation*, PLDI '15, pp. 521–532, 2015.

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

[14] W. Gropp, D. Kaushik, D. Keyes, and B. Smith, "Toward realistic performance bounds for implicit CFD codes," in *Parallel CFD '99*, 1999.

[15] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Parallel, Distributed and Network-Based Processing*, PDP '08, pp. 283–292, 2008.

[16] X. Li, M. J. Garzarán, and D. Padua, "A dynamically tuned sorting library," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*, (Washington, DC, USA), IEEE Computer Society, 2004.

[17] F. Franchetti, F. Mesmay, D. Mcfarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, (Berlin, Heidelberg), pp. 385–409, Springer-Verlag, 2009.

[18] E. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 135–158, Feb. 2004.

[19] A. El Zein and A. Rendell, "From sparse matrix to optimal gpu cuda sparse matrix vector product implementation," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pp. 808 –813, may 2010.

[20] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance optimization and modeling of blocked sparse kernels," *International Journal of High Performance Computing Applications*, vol. 21, no. 4, pp. 467–484, 2007.

[21] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, and P. Chen, "A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, (New York, NY, USA), pp. 2:1–2:8, ACM, 2011.

[22] R. Vuduc, J. Demmel, and J. Bilmes, "Statistical models for empirical search-based performance tuning," *Int. J. High Perform. Comput. Appl.*, vol. 18, pp. 65–94, Feb. 2004.

[23] C. Zheng, S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu, "Biell: A bisection ellpack-based storage format for optimizing spmv on {GPUs}," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2639 – 2647, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.

[24] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: Yet another spmv framework on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, (New York, NY, USA), pp. 107–118, ACM, 2014.

[25] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 781–792, IEEE Press, 2014.

[26] J. Choi, A. Singh, and R. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," in *Principles and Practice of Parallel Programming*, PPoPP '10, pp. 115–126, 2010.

[27] W. Abu-Sufah and A. Karim, "An effective approach for implementing sparse matrix-vector multiplication on graphics processing units," in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pp. 453–460, June 2012.

[28] K. Kourtis, G. Goumas, and N. Koziris, "Exploiting compression opportunities to improve spmxv performance on shared memory systems," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 16:1–16:31, Dec. 2010.

[29] F. Vzquez, J. J. Fernndez, and E. M. Garzn, "A new approach for sparse matrix vector product on nvidia gpus," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[30] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using cuda based on a hybrid sparse matrix format," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 11, pp. V11–161–V11–165, Oct 2010.

[31] F. Vazquez, G. Ortega, J. Fernandez, and E. Garzon, "Improving the performance of the sparse matrix vector product with gpus," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 1146–1151, June 2010.

[32] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient smvm kernels," in *Proc. of the 23rd Int. Conf. on Supercomputing*, ICS '09, pp. 100–109, ACM, 2009.

[33] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Supercomputing*, ICS '06, pp. 307–316, 2006.

[34] B. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Supercomputing*, ICS '12, pp. 353–364, 2012.

[35] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[36] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations," 1994.

[37] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, (New York, NY, USA), pp. 87–96, ACM, 2008.

[38] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, CSE '09, (Washington, DC, USA), pp. 247–256, IEEE Computer Society, 2009.

[39] U. Catalyurek and C. Aykanat, "Decomposing irregularly sparse matrices for parallel matrix-vector multiplication," in *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '96, (London, UK), pp. 75–86, Springer-Verlag, 1996.

[40] R. Geus and S. Rllin, "Towards a fast parallel sparse symmetric matrixvector multiplication," *Parallel Computing*, vol. 27, no. 7, pp. 883 – 896, 2001. Linear systems and associated problems.

[41] E. jin Im and K. Yelick, "Optimizing sparse matrix vector multiplication on smps," in *In Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[42] J. Pichel, D. Heras, J. Cabaleiro, and F. Rivera, "Improving the locality of the sparse matrix-vector product on shared memory multiprocessors," in *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pp. 66–71, Feb 2004.

[43] E.-J. Im and K. A. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, (London, UK), pp. 127–136, Springer-Verlag, 2001.

[44] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, (New York, NY, USA), ACM, 1999.

[45] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM J. Res. Dev.*, vol. 41, pp. 711–726, Nov. 1997.

[46] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, (Los Alamitos, CA, USA), pp. 1–35, IEEE Computer Society Press, 2002.

[47] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proceedings of the First International Conference on High Performance Computing and Communications*, HPCC'05, (Berlin, Heidelberg), pp. 807–816, Springer-Verlag, 2005.

[48] O. Temam and W. Jalby, "Characterizing the behavior of sparse algorithms on caches," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, (Los Alamitos, CA, USA), pp. 578–587, IEEE Computer Society Press, 1992.

[49] I. White, J.B. and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pp. 66–71, Dec 1997.

[50] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, pp. 419–422, June 1968.

[51] D. R. Engler, "Vcode: A retargetable, extensible, very fast dynamic code generation system," in *In PLDI 96: Proceedings Of The ACM SIGPLAN 1996 Conference On Programming Language Design And Implementation*, pp. 160–170, 1996.

[52] H. Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, USA, 1992. UMI Order No. GAX92-32050.

[53] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad, "Fast, effective dynamic compilation," *SIGPLAN Not.*, vol. 31, pp. 149–159, May 1996.

[54] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, "Optimistic incremental specialization: Streamlining a commercial operating system," *SIGOPS Oper. Syst. Rev.*, vol. 29, pp. 314–321, Dec. 1995.

[55] O. Agesen and U. Hölzle, "Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages," *SIGPLAN Not.*, vol. 30, pp. 91–107, Oct. 1995.

[56] C. Chambers and D. Ungar, "Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, (New York, NY, USA), pp. 146–160, ACM, 1989.

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

[58] J. Carette and O. Kiselyov, "Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code," *Sci. Comput. Program.*, vol. 76, pp. 349–375, May 2011.

[59] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi, "Building efficient query engines in a high-level language," *Proc. VLDB Endow.*, vol. 7, pp. 853–864, June 2014.

[60] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Proc. SciDAC, J. Physics: Conf. Ser.*, vol. 16, pp. 521–530, 2005.

[61] C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, no. 12, pp. 3–35, 2001.

[62] M. Frigo and S. G. Johnson, "The fastest fourier transform in the west," Tech. Rep. MIT/LCS/TR-728, Massachusetts Institute of Technology Laboratory for Computer Science, 1997.

[63] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *Proc. of the Int. Symp. on Code Generation and Optimization*, CGO '05, pp. 123–134, IEEE Computer Society, 2005.

[64] K. Stock, L.-N. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," *ACM Trans. Archit. Code Optim.*, vol. 8, pp. 50:1–50:23, Jan. 2012.

[65] A. Trouv, A. Cruz, H. Fukuyama, J. Maki, H. Clarke, K. Murakami, M. Arai, T. Nakahira, and E. Yamanaka, "Using machine learning in order to improve automatic simd instruction generation," *Procedia Computer Science*, vol. 18, no. 0, pp. 1292 – 1301, 2013. 2013 Int. Conf. on Computational Science.

[66] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *Parallel and Distributed Processing Symp.*, IPDPS '14, pp. 501–512, 2014.

[67] R. G. Grimes, D. R. Kincaid, and D. M. Young, "ITPACK 2.0 user's guide," Report CNA-150, Center for Numerical Analysis, University of Texas at Austin, Austin, TX, USA, Aug. 1978.

[68] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization*, CGO '04, pp. 75–86, 2004.

[69] "Llvm web site." http://llvm.cs.uiuc.edu, 2013.

[70] J. Byun, R. Lin, K. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," Tech. Rep. UCB/EECS-2012-215, EECS Department, U. of California, Berkeley, Nov 2012.

[71] "OpenMP API for parallel programming, version 3.0." http://openmp.org/wp, 2009.

[72] C. Consel, J. Lawall, and A. Le Meur, "A tour of tempo: A program specializer for the c language," *Sci. Comput. Program.*, vol. 52, no. 1-3, pp. 341–370, 2004.

[73] S. Kamin, L. Clausen, and A. Jarvis, "Jumbo: Run-time code generation for java and its applications," in *Code Generation and Optimization*, CGO '03, pp. 48–56, 2003.

[74] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," in *Generative Prog. and Component Engineering*, GPCE '10, pp. 127–136, 2010.

[75] M. Frigo, "A fast fourier transform compiler," in *Programming Language Design and Implementation*, PLDI '99, pp. 169–180, 1999.

[76] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[77] R. Vuduc, J. Demmel, and K. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conf. Series*, vol. 16, no. 1, p. 521, 2005.

[78] A. El Zein and A. Rendell, "Generating optimal cuda sparse matrixvector product implementations for evolving gpu hardware," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 1, pp. 3–13, 2012.

[79] W. Armstrong and A. Rendell, "Reinforcement learning for automated performance tuning," in *Cluster Computing*, pp. 411–420, Sept 2008.

[80] W. Armstrong and A. Rendell, "Runtime sparse matrix format selection," *Procedia Computer Science*, vol. 1, no. 1, pp. 135 – 144, 2010.

[81] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication," *SIGPLAN Not.*, vol. 48, pp. 117–126, June 2013.

[82] B. Neelima, G. R. M. Reddy, and P. S. Raghavendra, "Predicting an optimal sparse matrix format for spmv computation on gpu," in *Parallel & Distributed Processing Symp. Workshops*, IPDPSW '14, pp. 1427–1436, 2014.

[83] W. Abu-Sufah and A. Abdel Karim, "Auto-tuning of sparse matrix-vector multiplication on graphics processors," in *Supercomputing*, vol. 7905 of *Lecture Notes in Computer Science*, pp. 151–164, Springer, 2013.

[84] "Matrix Market Web Site." http://math.nist.gov/MatrixMarket, 1997.

[85] T. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[86] "Amd core math library user guide 6.0.6." http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/acml.pdf, 2013.

[87] F. Van Zee and R. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Trans. Math. Softw.*, vol. 41, pp. 14:1–14:33, June 2015.

[88] F. Van Zee, E. Chan, R. van de Geijn, E. Quintana-Ortí, and G. Quintana-Ortí, "The libflame library for dense matrix computations," *Computing in Science Engineering*, vol. 11, pp. 56–63, Nov 2009.

[89] 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, "Scikit-learn: Machine learning in Python," *J. of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[90] I. U. Akgün, *Performance Evaluation Of Unfolded Sparse Matrix-Vector Multiplication*. Master thesis, Ozyegin University, 2015.

[91] K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison, "Model-driven simd code generation for a multi-resolution tensor kernel," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1058–1067, May 2011.

[92] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[93] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing polynomial expressions by algebraic factorization and common subexpression elimination," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, pp. 2012–2022, Oct 2006.

[94] F. P. Russell and P. H. J. Kelly, "Optimized code generation for finite element local assembly using symbolic manipulation," *ACM Trans. Math. Softw.*, vol. 39, pp. 26:1–26:29, July 2013.

[95] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology," in *Proc. of the 11th Int. Conf. on Supercomputing*, ICS '97, pp. 340–347, ACM, 1997.

[96] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus," *IEEE Trans. on Parallel and Distributed Systems*, vol. 25, pp. 1112–1123, May 2014.

[97] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, and P. Chen, "A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, (New York, NY, USA), pp. 2:1–2:8, ACM, 2011.

[98] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus: Implications for graph mining," *Proc. VLDB Endow.*, vol. 4, pp. 231–242, Jan. 2011.

[99] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for spmv on gpu using probabilistic modeling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, pp. 196–205, Jan 2015.

[100] D. Grewe and A. Lokhmotov, "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, pp. 12:1–12:8, 2011.

[101] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proc. of the IEEE*, vol. 93, pp. 293–312, Feb 2005.

[102] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "Poet: Parameterized optimizations for empirical tuning," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–8, March 2007.

[103] J. A. Gunnels, G. M. Henry, and R. A. v. d. Geijn, "A family of high-performance matrix multiplication algorithms," in *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, (London, UK), pp. 51–60, Springer-Verlag, 2001.

[104] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, (Washington, DC, USA), pp. 169–176, IEEE Computer Society, 2004.

[105] P. Guo and L. Wang, "Auto-tuning cuda parameters for sparse matrix-vector multiplication on gpus," in *Computational and Information Sciences (ICCIS), 2010 International Conference on*, pp. 1154–1157, Dec 2010.

[106] I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based gpus," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–12, May 2012.

[107] X. Li, M. J. Garzaran, and D. Padua, "Optimizing sorting with genetic algorithms," in *Proc. of the Int. Symp. on Code Generation and Optimization*, CGO '05, pp. 99–110, IEEE Computer Society, 2005.

[108] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristicsx," in *Proc. of the 10th Int. Conf. on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, pp. 41–50, Springer-Verlag, 2002.

[109] "Open Reearch Compiler," 2009. http://ipf-orc.sourceforge.net.

[110] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 185–197, IEEE Computer Society, 2007.

[111] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 2009.

[112] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," *SIGPLAN Not.*, vol. 41, pp. 229–240, Oct. 2006.

[113] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.

[114] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 4:1–4:12, IEEE Press, 2008.

[115] G. Murthy, M. Ravishankar, M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–11, April 2010.

[116] T. Kisuki, P. Knijnenburg, and M. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Parallel Architectures and Compilation Techniques, 2000. Proc.. Int. Conf. on*, pp. 237–246, 2000.

[117] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–11, May 2009.

[118] J. Morlan, S. Kamil, and A. Fox, "Auto-tuning the matrix powers kernel with sejits," in *High Performance Computing for Computational Science - VECPAR 2012* (M. Dayd, O. Marques, and K. Nakajima, eds.), vol. 7851 of *Lecture Notes in Computer Science*, pp. 391–403, Springer Berlin Heidelberg, 2013.

[119] B. Catanzaro, S. A. Kamil, Y. Lee, K. Asanovi, J. Demmel, K. Keutzer, J. Shalf, K. A. Yelick, and A. Fox, "Sejits: Getting productivity and performance with selective embedded jit specialization," Tech. Rep. UCB/EECS-2010-23, EECS Department, University of California, Berkeley, Mar 2010.

[120] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 10:1–10:12, IEEE Computer Society Press, 2012.

[121] "Insieme Compiler and Runtime Infrastructure," 2001. http://insieme-compiler.org.

[122] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, (Los Alamitos, CA, USA), pp. 1–11, IEEE Computer Society Press, 2002.

[123] I.-H. Chung and J. K. Hollingsworth, "Using information from prior runs to improve automated tuning systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 30–, IEEE Computer Society, 2004.

[124] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Programming Language Design and Implementation*, PLDI '15, pp. 379–390, 2015.

[125] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao, "Optimizing spmv for diagonal sparse matrices on gpu," in *Parallel Processing*, ICPP '11, pp. 492–501, 2011.

[126] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, "Next-generation generic programming and its application to sparse matrix computations," in *Supercomputing*, ICS '00, pp. 88–99, 2000.

[127] P. Lee and M. Leone, "Optimizing ml with run-time code generation," in *Programming Language Design and Implementation*, PLDI '96, pp. 137–148, 1996.

[128] B. Aktemur, Y. Kameyama, O. Kiselyov, and C. Shan, "Shonan challenge for generative programming," in *Partial Evaluation and Program Manipulation*, PEPM '13, pp. 147–154, 2013.

[129] C. Consel and F. Noël, "A general approach for run-time specialization and its application to c," in *Principles of Programming Languages*, POPL '96, pp. 145–156, 1996.

[130] F. Gustavson, W. Liniger, and R. Willoughby, "Symbolic generation of an optimal crout algorithm for sparse systems of linear equations," *J. ACM*, vol. 17, pp. 87–109, Jan. 1970.

[131] Y. Fukui, H. Yoshida, and S. Higono, "Supercomputing of circuits simulation," in *Supercomputing*, SC '89, pp. 81–85, 1989.

[132] P. Giorgi and B. Vialla, "Generating optimized sparse matrix vector product over finite fields," in *Mathematical Software*, ICMS '14, pp. 685–690, 2014.

[133] G. Belter, E. Jessup, I. Karlin, and J. Siek, "Automating the generation of composed linear algebra kernels," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pp. 1–12, Nov 2009.

[134] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 311–320, ACM, 2012.

[135] F. A. Kamil Shoaib, Coetzee Derrick, "Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization," in *Proceedings of the 10th Python in Science Conference*, SciPy 2011, 2011.

[136] J. Shin, M. Hall, J. Chame, C. Chen, and P. Hovland, "Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology," in *Software Automatic Tuning* (K. Naono, K. Teranishi, J. Cavazos, and R. Suda, eds.), pp. 353–370, Springer New York, 2010.

[137] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," tech. rep., University of Southern California, 2008.

[138] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up nek5000 with autotuning and specialization," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 253–262, ACM, 2010.

[139] M. Christen, O. Schenk, and H. Burkhart, "Automatic code generation and tuning for stencil kernels on modern shared memory architectures," *Comput. Sci.*, vol. 26, pp. 205–210, June 2011.

[140] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Languages and Compilers for Parallel Computing* (G. Gao, L. Pollock, J. Cavazos, and X. Li, eds.), vol. 5898 of *Lecture Notes in Computer Science*, pp. 50–64, Springer Berlin Heidelberg, 2010.

[141] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," *SIGPLAN Not.*, vol. 44, pp. 38–49, June 2009.

[142] D. Han, S. Xu, L. Chen, and L. Huang, "Pads: A pattern-driven stencil compiler-based tool for reuse of optimizations on gpgpus," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, (Washington, DC, USA), pp. 308–315, IEEE Computer Society, 2011.

[143] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, (Washington, DC, USA), pp. 879–892, IEEE Computer Society, 2011.

[144] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth, "Parallel parameter tuning for applications with performance variability," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 57–, IEEE Computer Society, 2005.

VITA

Buse Yılmaz obtained her B.S. degree and M.S. degree from Yeditepe University, Department of Computer Engineering in 2009 and 2011. Her research interests include runtime program generation, compilers, parallel computing, high performance computing, and autotuning. She is also interested in algorithms and programming languages.