PERFORMANCE EVALUATION OF UNFOLDED SPARSE MATRIX-VECTOR MULTIPLICATION

A Thesis

by

İbrahim Ümit Akgün

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

Master

in the
Department of Computer Science

Özyeğin University
January 2015

PERFORMANCE EVALUATION OF UNFOLDED SPARSE MATRIX-VECTOR MULTIPLICATION

Approved by:

Assistant Professor Barış Aktemur, Advisor
Department of Computer Science
Özyeğin University

Associate Professor Fatih Uğurdağ
Department of Electrical and Electronics Engineering
Özyeğin University

Assistant Professor Furkan Kıraç
Department of Computer Science
Özyeğin University

Date Approved: 15 October 2014

To My Family

ABSTRACT

Sparse matrix-vector multiplication (spMV) is a kernel operation in scientific computation. There exist problems where a matrix is repeatedly multiplied by many different vectors. For such problems, specializing the spMV code based on the matrix has the potential of producing significantly faster code. This, in fact, has been one of the motivational examples of program generation. Using program generation, spMV code can be unfolded fully to eliminate loop overheads as well as enable high-impact optimizations. In this work we focus on specialization of spMV by unfolding the code according to a given matrix. We provide an experimental evaluation of performance using 70 sparse matrices collected from real-world scientific computation domains. We present optimizations with which high-performant assembly code can be generated rapidly without having to generate source-level code and go through all the phases of a general-purpose compiler. We finally present how one of the optimizations we studied can be implemented as a code-transforming pass.

ÖZETÇE

Seyrek matris-vektör çarpımı (spMV) bilimsel hesaplamalarda kullanılan çok temel bir işlemdir. Kimi bilimsel problemlerde aynı matris farklı vektörlerle tekrar tekrar çarpılmaktadır. Bu problemlerde kullanılan spMV kodunu matrise göre özelleşmiş bir şekilde optimize edersek çok ciddi performans artışları sağlanabilir. Bunu gerçekleştirmek için program üretimi teknikleri uygundur. Program üretimi ile spMV kodundaki döngü yükleri kaldırılabilir, ayrıca etkili eniyilemeler uygulanabilir. Bu çalışmada, spMV kodunun tam döngü açılımı vasıtasıyla çarpımı yapılmak istenen matrise göre özelleştirilmesini inceledik. Gerçek örneklerden oluşan 70 adet matris üzerinde deneysel performans çalışmaları yaptık. Ayrıca, kaynak kod üretimi ve sonrasında genel amaçlı derleyici kullanımına gerek bırakmayacak kadar yüksek kaliteli makine kodunu hızlı bir şekilde üretmemizi sağlayacak eniyilemeler sunuyoruz. Son olarak da, tanımladığımız eniyilemelerden birinin kod dönüşümü şeklinde nasıl tanımlanabileceğini gösteriyoruz.

ACKNOWLEDGMENTS

I would like to thank Assistant Professor Barış Aktemur for his support and advices.And my special thanks go to my family for supporting me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Sparse matrix-vector multiplication (**spMV**) is a kernel operation used intensively in many scientific domains such as finite element modeling, circuit design, simulation, etc. The real-world use-cases of spMV usually involve a large number of multiplications with big matrices. Therefore, optimizing spMV is desirable and has a wide impact.

A major factor in the performance of spMV is memory [1, 2]. SpMV usually suffers from the CPU-memory bottleneck: CPU waits for data to arrive from the memory. Well-known techniques such as hardware/software prefetching [3] fall short in fixing the problem because the sparsity of the matrix in spMV creates irregular memory access patterns when the matrix's elements are spread out in non-regular ways.

Optimization of spMV has been studied extensively; a complete overview of the literature, even if it were possible, would be out of the scope of this work. Previous approaches in general focus on reducing the CPU-memory bottleneck by improving the use of the CPU cache and/or reducing the amount of data required for the operation. To this aim, several matrix data representations and custom optimizations have been investigated to make spMV more efficient for various targets such as modern multicore CPUs and GPUs [4, 5, 6, 7]. Among these approaches, generative programming aims to optimize spMV by specializing the multiplication for a given matrix [8]. In this approach, specialization helps reduce the number of executed instructions as well as improve the memory access patterns.

Generative approaches are particularly useful for the problems where the same matrix is multiplied with many different vectors (so that the cost of specialization pays off). This is the case seen in, for instance, the so-called Krylov subspace problems where iterative methods like conjugate gradient or generalized minimal residual method (GMRES) are used. In these contexts, a sparse matrix (with fixed values, or variable values but fixed non-zero positions) is multiplied with several hundreds of vectors; the exact number of iterations depends on the parameters of the actual problem, such as the desired accuracy and the matrix preconditioner [9].

In [8], several generative methods have been investigated to address the optimization of spMV. One of these methods is to unfold the spMV loop. This method was also recently formulated as a Shonan Challenge in the context of Hidden Markov Modeling [10]. A drawback of unfolding is that the produced code may become too long. This, in return, may have a negative impact on the instruction-cache behaviour.

1.1 Contributions

In this dissertation, we focus on unfolding the spMV loop. We make the following contributions:

- Based on experiments, we show that unfolding not always gives speedup. Therefore, although it provides a simple and easy-to-explain example to motivate program generation, it should be used with a grain of salt for large, real-world matrices.

- We examine five low-level (i.e. at the machine instruction level) optimizations that aim to increase the speed of spMV code. In four of the five optimizations, we observe performance improvements. We argue that by performing these optimizations on a straightforwardly generated code, it is possible to achieve the performance of code that is generated by an industry-level compiler. This way, time-taking analyses and transformations may be by-passed, and code can be generated much rapidly.

- The optimizations we study are not dependent on any particular matrix; they can be integrated into compilers. As a proof of concept, we provide an implementation for one of the optimizations as a pass in the LLVM compiler infrastructure [11, 12]. Being able to define optimizations modularly is crucial for their reusability.

A major obversation we make in this thesis is that reducing the code size has substantial impact on the performance. Code size reduction optimizations are particularly important for embedded devices with limited-storage [13]. Those optimizations consider a wide range of code in general. In this thesis, we have focused on low-level investigation of the code that is the outcome of unfolding. In-depth performance evaluation of other specialization methods is left as a future work. The optimizations we studied are appropriate for low-level code generation after higher-level transformations such as those in [14] are considered.

1.2 Sparse Matrix-Vector Multiplication

Sparse matrices are the matrices that contain a large number of zero elements (e.g. 90%). While a dense matrix is typically stored as a two-dimensional array, sparse matrices are stored in custom formats that provide space savings. *Condensed sparse row*, abbreviated as CSR, is such a format.

CSR format represents a matrix using three arrays, as shown in Figure 1:

- `vals` array contains the non-zero values of the matrix in row-major order.

- `cols` array contains the column indices of the non-zero values stored in the `vals` array.

- `rows` array contains, for each row of the matrix, the starting index of the elements of the row in the `vals` array.

$$
\begin{pmatrix}
\cdot & \cdot & 0.1 & \cdot & 0.2 & \cdot & 0.3 & \cdot \\
\cdot & 0.4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0.5 & 0.6 \\
0.7 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 0.8 & 0.9 & \cdot & \cdot & \cdot & \cdot & \cdot \\
1.0 & 1.1 & \cdot & 1.2 & 1.3 & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1.4 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1.5 & \cdot & \cdot & \cdot & \cdot
\end{pmatrix}
$$

$$\texttt{rows} = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0 & 3 & 4 & 6 & 7 & 9 & 13 & 14 & 15 \\ \hline\end{array}$$

$$\texttt{vals} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 & 0.7 & 0.8 & 0.9 & 1.0 & 1.1 & 1.2 & 1.3 & 1.4 & 1.5 \\ \hline\end{array}$$

$$\texttt{cols} = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 2 & 4 & 6 & 1 & 6 & 7 & 0 & 1 & 2 & 0 & 1 & 3 & 4 & 2 & 3 \\ \hline\end{array}$$

Figure 1: A sample matrix and its representation in the CSR format.

```
for (int i = 0; i < N; i++) {
   double ww = 0.0;
   for (int k = rows[i]; k < rows[i+1]; k++) {
      ww += vals[k] * v[cols[k]];
   }
   w[i] += ww;
}
```

Figure 2: SpMV implementation that computes $\texttt{w} \leftarrow \texttt{w} + M\texttt{v}$, where M is represented using the `rows`, `cols`, and `vals` arrays according to the CSR format. This code will be refered as `PlainSpMV`.

Assuming that the dimension of the matrix is $N \times N$, and it has NZ non-zero elements, the `vals` and `cols` arrays are of length NZ, whereas the `rows` array is of length $N+1$. The last element in the `rows` array (i.e. the "+1"), denotes the end position of the last row. Figure 1 is an example of the CSR representation.

Based on the CSR format, a sparse matrix-vector multiplication is implemented as given in Figure 2, where `v` is the input vector (assumed to be of length `N`), and `w` is the output vector (again, assumed to be of length `N`). The code calculates $\texttt{w} \leftarrow \texttt{w} + M\texttt{v}$, where M is represented using the `rows`, `cols`, and `vals` arrays.

In this dissertation we always assume that the matrix elements are double-precision floating point numbers, and all the matrices are square.

1.3 Unfolding SpMV

Program generation is based on the idea that if a subset of the inputs to a program are available, the program can be optimized using the available information until the rest of the inputs become available. Many iterative problems in scientific computation require the multiplication of a single matrix with many different vectors [9]. That is, the matrix is static while the input vector is dynamic. Then, the `PlainSpMV` code given in Figure 2 can be optimized with respect to the matrix, so that anytime the matrix is to be multiplied with a vector, the optimized version can be used for faster execution.

The most straightforward optimization would be to unfold the for-loops in Figure 2. (For other possible methods, see [8].) A variation of unfolding in the context of Hidden Markov Models was also formulated as a Shonan Challenge [10]. Unfolding of vector-vector multiplication is a well-known motivational example in the area of program generation [15].

Unfolding the code for the matrix of Figure 1 gives the code in Figure 3. With unfolding, we basically obtain a statement for each row of the matrix. From now on, we will refer to this method of specializing the `PlainSpMV` code as `Unfolding`.

Unfolding the loop frees us from the overheads of the loop. The elements of the input array `v` are no longer referenced indirectly via the `cols` array; instead, the indices of `v` are embedded in the code. The drawback of unfolding is that it results in very long code for large matrices. When the executed code is this long, some compilers may give up on some of the optimizations that they would otherwise perform. Also, the misses in the instruction cache may remarkably decrease efficiency.

```
w[0] += 0.1 * v[2] + 0.2 * v[4] + 0.3 * v[6];
w[1] += 0.4 * v[1];
w[2] += 0.5 * v[6] + 0.6 * v[7];
w[3] += 0.7 * v[0];
w[4] += 0.8 * v[1] + 0.9 * v[2];
w[5] += 1.0 * v[0] + 1.1 * v[1] + 1.2 * v[3] + 1.3 * v[4];
w[6] += 1.4 * v[2];
w[7] += 1.5 * v[3];
```

Figure 3: Unfolding the loops in Figure 2 for the matrix in Figure 1. This way of unfolding will be refered as `Unfolding`.

1.3.1 Remarks on the Performance of Unfolded Code

We have observed that `Unfolding` makes a big difference in the performance if the matrix has few distinct nonzero values (we show benchmarking results in Section 1.5). Here we shed some light on why this is the case.

Suppose that after unfolding we have the following two statements in the code:

```
w[10] += 0.9*v[2] + 0.9*v[4] + 0.5*v[7] + 0.3*v[8];

w[11] += 0.5*v[7] + 0.3*v[8] + 0.9*v[14];
```

In the object code produced by the compiler, the matrix values are emitted to the data section of the code, and loaded from there using a move instruction, just like reading values from an array (we have verified this for icc, gcc, and clang). Therefore, the code given above is essentially equivalent to putting the matrix values into an array, and reading the values from there. However, because the matrix values appear as constants in the code, instead of blindly emitting the values into the data section, the compiler may emit the distinct nonzero values only. This creates a pool from where values are retrieved. So, the compiler may output object code as if the source code were as given below. This optimization significantly reduces the loads from the memory if the distinct nonzero values are few.

```
double M[7] = {0.9, 0.5, 0.3};
w[10] += M[0]*v[2] + M[0]*v[4] + M[1]*v[7] + M[2]*v[8];
w[11] += M[1]*v[7] + M[2]*v[8] + M[0]*v[14];
```

Furthermore, because the nonzero values are constants, and, again, if the number of distinct values are few, the compiler may find opportunities for arithmetic optimizations. A potential optimization is the reverse distribution of multiplication over addition; i.e. $c \times a + c \times b = c \times (a + b)$. When applied, this optimization reduces the number of floating point multiplication instructions.

Another optimization is to eliminate multiplication when the nonzero value is the identity element 1, i.e. $1 \times a = a$. This further reduces the number of multiplications. A similar optimization is to emit a subtraction instruction for $-1 \times a$, instead of multiplication.

Yet another potential optimization is common subexpression elimination (CSE). When the distinct values are few, there may be many cases where an expression is detected in many places. The expression `M[1]*v[7] + M[2]*v[8]` in the code above is an example of this.

After applying CSE and arithmetic optimizations, the compiler would be able to emit code equivalent to the following:

```
double M[7] = {0.9, 0.5, 0.3};
double temp = M[1]*v[7] + M[2]*v[8];
w[10] += M[0]*(v[2] + v[4]) + temp;
w[11] += temp + M[0]*v[14];
```

When the distinct nonzero values are not few, applying CSE and creating a pool of distinct values may have negative impact on the performance because these transformations change the order of the memory addresses loaded. This may reduce the cache utilization. In the original `PlainSpMV` code (Figure 2), the values of the matrix

```
w[0] += vals[0] * v[2] + vals[1] * v[4] + vals[2] * v[6];
w[1] += vals[3] * v[1];
w[2] += vals[4] * v[6] + vals[5] * v[7];
w[3] += vals[6] * v[0];
w[4] += vals[7] * v[1] + vals[8] * v[2];
w[5] += vals[9] * v[0] + vals[10] * v[1]
      + vals[11] * v[3] + vals[12] * v[4];
w[6] += vals[13] * v[2];
w[7] += vals[14] * v[3];
```

Figure 4: Unfolding the loops in Figure 2 according to the *positions* of the matrix in Figure 1; this version will be refered to as `UnfoldingV2`.

are loaded in strict consecutive order, giving as good cache utilization as possible for the `vals` array.

1.4 Unfolding According to the Matrix Pattern

Unfolding, as shown in Figure 3, uses all the matrix data. So, to be able to unfold the code, all the matrix data have to be available. There exist, however, *pattern matrices* in the scientific computing area. Pattern matrices are those where the positions of the non-zero values are known, but the actual values are not determined yet. There are also scientific problems where the values of elements of a matrix change from one iteration to the other, while the positions of these elements stay the same. For these cases, it may be preferable to unfold the `PlainSpMV` code according to the positions of nonzeros without embedding the actual values in the code. In this case, the values are read from the `vals` array. Figure 4 shows the obtained code when unfolding is done in this manner for the matrix in Figure 1. From this point on, we refer to this version of unfolding as `UnfoldingV2`.

1.5 Performance Evaluation of `Unfolding` and `UnfoldingV2`

We generated code using `Unfolding` and `UnfoldingV2` methods for 70 matrices arbitrarily selected from the Matrix Market [16] and the University of Florida collection

[17]. All the matrices are sparse and square. They are used in a variety of real world applications. The number of nonzero element of these matrices vary between ~2000 and 50,000. In our environment, unfolding rarely brings speedup for matrices that have more than 50,000 elements; therefore we did not include these matrices in our study. Our matrix set contains pattern matrices; these are annotated with a (p) mark next to their name.

In our experiment, we generated source code for each matrix using the `Unfolding` and `UnfoldingV2` approaches; in addition, there is the `PlainSpMV` code. Source codes are then compiled using the icc compiler version 14.0 with the `-O3 -no-vec` flags (i.e. vectorization is turned off). Then, we executed the three versions of multiplication for each matrix for several thousand times, measured the elapsed time, repeated this for 5 times, and used the minimum of those times. How many times to execute the multiplication function was determined according to the matrix size: If the number of nonzero values is less than 5,000, we repeated the functions for 500,000 times; if there are more than 5,000 but fewer than 10,000 elements, the number of iterations was 200,000; for matrices that have more than 10,000 elements, we repeated the code for 100,000 times. We determined these number of iterations so that a run for each matrix is about 2 seconds or more; this allows for reliable time measurements with negligible noise. All the code was ran single-threaded. The experiment was done on an unloaded machine that has an Intel Xeon E5-2620 2.00 GHz CPU with 32K L1 I/D cache, 256K L2 cache, 15M L3 cache. Before each multiplication, we zero-out the output vector `w`. The source of pattern matrices, which are annotated with a (p) next to their name, do not contain any values; for these matrices we generated nonzero values where all the values are different from each other.

The speedups obtained by unfolding with respect to `PlainSpMV` are given in Tables 1 and 2. For each matrix, we list the number of rows, number of nonzero values, number of distinct values, the time it took to run `PlainSpMV` (in microseconds), the ratio of `PlainSpMV` time to `Unfolding`'s time, and finally the ratio of `PlainSpMV` to `UnfoldingV2`'s time. Having a ratio larger than 1 means that unfolding is faster than `PlainSpMV`. These cases are marked in **bold font** in the tables. The tables are sorted in ascending order according to the number of nonzero values.

Based on the data given in Tables 1 and 2, `Unfolding` gives an average performance of 1.46x the performance of `PlainSpMV` for 70 matrices. In 50 of these 70 matrices we see a speedup; in 20 there is slowdown. For all the matrices where the ratio of distinct values to the total number of nonzeros is less than 1%, there is speedup. These matrices are: olm5000, gr_30_30, saylr4, G33, rdb1250, cdde3, gre_1107, jpwh_991, M80PI_n1, rw5151, and orsreg_1. In fact, for these matrices, the performance of `Unfolding` is remarkably good: 2.26x on the average.

It is not surprising that `UnfoldingV2` gives worse performance than `Unfolding`. On the average, `UnfoldingV2` performs 1.03x the performance of `PlainSpMV`. This time, speedup is observed for only 29 matrices out of 70. However, the most striking remark about `UnfoldingV2` is that for all the pattern matrices (there are 17) except lshp3466 and dwt_2680, `UnfoldingV2` gives speedup with respect to `PlainSpMV`. For pattern matrices, the average performance is 1.59x.

Our conclusion from this experiment is that unfolding as a specialization method for spMV does not necessarily give speedup; it should not be applied blindly. That said, significant speedup can be expected when there are few distinct values in the matrix. Finally, `UnfoldingV2`, i.e. unfolding according to the matrix pattern, usually provides substantial speedup for pattern matrices.

Matrix	N	NZ	Distinct values	PlainSpMV time (μs)	Unfolding time wrt PlainSpMV	UnfoldingV2 time wrt PlainSpMV
dwt_419 (p)	419	1991	1991	5.8	**1.72**	**1.73**
str__600	363	3279	1972	6.1	**1.29**	**1.15**
minnesota (p)	2642	3303	3303	24.5	**2.68**	**2.99**
bcspwr06 (p)	1454	3377	3377	19.0	**2.69**	**2.70**
west0989	989	3518	1776	10.5	**2.02**	**1.62**
bfw398a	398	3678	92	5.9	**1.23**	0.89
bcsstk19	817	3835	1852	7.4	**1.21**	**1.14**
bcspwr08 (p)	1624	3837	3837	21.6	**2.69**	**2.71**
ck656	656	3884	3054	6.2	0.98	0.94
can__634 (p)	634	3931	3931	7.7	**1.16**	**1.10**
tub1000	1000	3996	1990	7.4	**1.24**	**1.09**
G33	2000	4000	2	13.0	**2.37**	**1.25**
bcsstk06	420	4140	1045	6.4	**1.02**	0.96
hor__131	434	4182	1553	6.4	**1.06**	0.88
gr_30_30	900	4322	2	8.0	**4.04**	**1.01**
pde900	900	4380	3248	8.1	**2.74**	0.94
cdde3	961	4681	5	8.7	**3.50**	0.84
bp__1600	822	4841	1803	13.2	**1.86**	**1.46**
email (p)	1133	5451	5451	18.3	**1.55**	**1.66**
steam2	600	5660	1071	8.6	0.93	0.88
gre_1107	1107	5664	11	14.5	**1.82**	**1.21**
fs_760_1	760	5739	4743	9.2	**1.50**	0.78
dwt_1242 (p)	1242	5834	5834	14.0	**1.66**	**1.21**
e05r0000	236	5846	1269	6.4	0.68	0.64
fpga_dcop_51	1220	5892	953	14.5	**1.63**	**1.36**
jpwh_991	991	6027	14	14.5	**2.14**	**1.02**
EVA (p)	8497	6726	6726	37.3	**1.77**	**1.76**
can_1072 (p)	1072	6758	6758	15.5	**1.19**	**1.12**
rdb1250	1250	7300	6	12.6	**1.45**	0.80
west2021	2021	7310	4235	21.2	**1.61**	**1.22**
mahindas	1258	7682	3291	13.9	**1.28**	0.85
GD06_Java (p)	1538	8032	8032	23.2	**1.28**	**1.24**
nos3	960	8402	149	11.9	**1.04**	0.71
blckhole (p)	2132	8502	8502	20.5	1.00	**1.02**
c-18	2169	8657	4861	18.7	**1.42**	**1.06**

Table 1: (Part 1 of 2) Speedups obtained by `Unfolding` and `UnfoldingV2`, with respect to `PlainSpMV`'s performance.

Matrix	**N**	**NZ**	**Distinct values**	**PlainSpMV time (μs)**	**Unfolding time wrt PlainSpMV**	**UnfoldingV2 time wrt PlainSpMV**
tols4000	4000	8784	3188	23.3	**2.61**	0.96
pores_2	1224	9613	5407	14.5	0.80	0.64
spiral	1434	9831	3089	15.6	**1.08**	0.63
M80PI_n1	4028	9927	70	25.8	**2.11**	0.86
dw2048	2048	10114	693	23.8	**2.02**	0.85
watt__1	1856	11360	6524	24.5	0.88	0.84
watt__2	1856	11550	6589	24.4	0.83	0.84
bayer09	3083	11767	5003	28.8	**1.09**	0.86
Pd	8081	13036	432	84.2	**2.80**	**1.78**
add20	2395	13151	7390	32.4	**1.04**	0.89
lshp3466 (p)	3466	13681	13681	34.8	0.91	0.91
dwt_2680 (p)	2680	13853	13853	34.1	0.93	0.93
as-735 (p)	7716	13895	13895	87.3	**1.83**	**1.84**
orsreg_1	2205	14133	111	24.3	**2.35**	0.63
ca-GrQc (p)	5242	14496	14496	63.0	**1.37**	**1.37**
adder_trans_02	1814	14579	10327	30.0	0.85	0.78
bcsstk26	1922	16129	13480	32.3	0.84	0.79
plat1919	1919	17159	17120	27.6	**1.06**	0.61
wang2	2903	19093	1727	34.3	**1.57**	0.64
coater1	1348	19457	1380	27.2	0.69	0.52
add32	4960	19848	13883	62.6	**1.17**	**1.07**
olm5000	5000	19996	6	38.2	**1.10**	0.71
rw5151	5151	20199	150	41.1	**2.25**	0.63
sherman5	3312	20793	15096	36.5	0.70	0.69
saylr4	3564	22316	11	45.8	**1.68**	0.72
Oregon-1 (p)	11492	23409	23409	136.4	**1.65**	**1.64**
mcfe	765	24382	24381	26.4	0.41	0.40
lnsp3937	3937	25407	4176	42.9	0.69	0.63
fidap002	441	26831	11118	26.1	0.36	0.35
bcsstk14	1806	32630	14044	45.0	0.56	0.54
cavity05	1182	32632	3280	36.7	0.47	0.42
p2p-Gnutella04 (p)	10879	39994	39994	134.7	**1.05**	**1.05**
mbeause	496	41063	2100	41.3	0.44	0.34
cry10000	10000	49699	49599	88.7	**1.96**	0.62
mbeaflw	496	49920	19778	48.4	0.33	0.33

Table 2: (Part 2 of 2) Speedups obtained by `Unfolding` and `UnfoldingV2`, with respect to `PlainSpMV`'s performance.

CHAPTER II

LOW-LEVEL OPTIMIZATIONS

In Chapter 1 we have shown how unfolding can be done at the source level. Once generated, the code is fed into a compiler so that the compiler can apply optimizations and finally convert the code into machine instructions by running analyses such as register allocation and instruction selection. Considering that code generation will be performed at runtime in many cases, executing all the phases of a compiler (from parsing at the front-end down to native code generation at the back-end) is a costly operation.

In Tables 3 and 4, we show how much time is spent to compile the source codes generated using the `Unfolding` and `UnfoldingV2` methods. Here, the codes are compiled using icc with the `-O3 -no-vec` flags. Compilation times have been measured using the `time` command. We report the measured time, and also the ratio of this time to the time of executing `PlainSpMV` once. The latter value is provided to give an impression of how many times we could have multiplied the matrix until the generated code becomes available; it is an underestimation because we do not include the cost of generating the source codes; only the compilation times are reported.

Generating the code at runtime using the naive "produce a source file and feed it to the compiler" approach is not feasible for unfolded spMV because the generated code is long and the compilation takes very long time. While an spMV multiplication is done in the orders of microseconds, compilation takes time in the order of seconds. Therefore, we ask the question "How rapidly can we generate code at runtime?" To this end, we have written a purpose-built compiler that takes a matrix as an input and produces a dynamically loadable object file at runtime. We use the LLVM [11, 12]

Matrix	PlainSpMV (μs)	Unfolding compilation (s)	UnfoldingV2 compilation (s)	Unfolding/ PlainSpMV	UnfoldingV2/ PlainSpMV
add20	32.4	3.49	4.41	107746	136235
add32	62.6	4.94	6.23	78927	99446
adder_trans_02	30.0	5.06	6.33	168907	211042
as-735 (p)	87.3	4.67	5.81	53472	66522
bayer09	28.8	3.54	4.24	122928	147340
bcspwr06 (p)	19.0	1.25	1.54	65467	80927
bcspwr08 (p)	21.6	2.15	2.57	99314	119038
bcsstk06	6.4	1.25	1.14	193841	177079
bcsstk14	45.0	8.44	9.56	187433	212353
bcsstk19	7.4	0.95	1.09	127306	146550
bcsstk26	32.3	3.75	4.31	115941	133278
bfw398a	5.9	0.98	1.26	167298	215268
blckhole (p)	20.5	3.87	2.63	188467	127984
bp__1600	13.2	1.44	1.93	109114	146753
c-18	18.7	1.69	2.33	90674	124523
ca-GrQc (p)	63.0	4.35	5.36	68994	85060
can__634 (p)	7.7	1.44	1.16	187448	150584
can_1072 (p)	15.5	1.59	1.93	103035	124767
cavity05	36.7	12.21	13.19	332579	359418
cdde3	8.7	0.53	1.76	60345	201875
ck656	6.2	0.91	1.08	147032	174379
coater1	27.2	6.24	7.72	229343	283893
cry10000	88.7	3.89	13.81	43897	155770
dw2048	23.8	2.53	3.80	106382	159552
dwt_1242 (p)	14.0	1.64	1.81	117146	129518
dwt_2680 (p)	34.1	6.67	3.84	195752	112768
dwt_419 (p)	5.8	0.56	0.67	95332	115014
e05r0000	6.4	2.17	2.36	338498	368944
email (p)	18.3	1.66	1.96	90878	107336
EVA (p)	37.3	2.78	3.71	74299	99279
fidap002	26.1	13.76	16.17	527212	619641
fpga_dcop_51	14.5	1.52	1.79	104411	123176
fs_760_1	9.2	1.31	2.00	142125	216881
G33	13.0	1.46	1.82	112279	139889

Table 3: (Part 1 of 2) The time it takes to compile generated code for each matrix using the `Unfolding` and `UnfoldingV2` methods.

Matrix	PlainSpMV (μs)	Unfolding compilation (s)	UnfoldingV2 compilation (s)	Unfolding/ PlainSpMV	UnfoldingV2/ PlainSpMV
GD06_Java (p)	23.2	2.59	2.81	111654	121224
gr_30_30	8.0	0.46	1.43	56675	177980
gre_1107	14.5	1.49	1.82	102209	124755
hor__131	6.4	1.18	1.35	184698	211240
jpwh_991	14.5	1.51	2.49	104341	172221
lnsp3937	42.9	5.80	6.76	135285	157852
lshp3466 (p)	34.8	6.21	4.28	178706	122991
M80PI_n1	25.8	2.15	3.57	83359	138504
mahindas	13.9	2.06	2.95	148849	212631
mbeaflw	48.4	37.15	44.36	767280	916011
mbeause	41.3	24.78	36.61	599859	886135
mcfe	26.4	9.66	11.22	365681	424803
minnesota (p)	24.5	1.44	1.85	58701	75252
nos3	11.9	2.11	2.27	177505	190442
olm5000	38.2	3.76	5.92	98353	155097
Oregon-1 (p)	136.4	8.28	10.29	60682	75487
orsreg_1	24.3	1.70	4.58	69774	188545
p2p-Gnutella04 (p)	134.7	8.48	10.91	62993	81030
Pd	84.2	18.59	27.02	220654	320689
pde900	8.1	0.52	1.68	64948	207982
plat1919	27.6	3.71	5.07	134236	183263
pores_2	14.5	2.18	2.60	150205	179199
rdb1250	12.6	1.72	2.44	136593	194064
rw5151	41.1	2.05	8.04	49833	195828
saylr4	45.8	5.31	7.08	115801	154474
sherman5	36.5	6.05	8.16	165780	223506
spiral	15.6	12.17	4.25	780822	272723
steam2	8.6	1.49	1.66	173358	192814
str__600	6.1	0.85	1.03	139767	169929
tols4000	23.3	1.69	3.87	72823	166410
tub1000	7.4	0.88	1.19	117798	159302
wang2	34.3	3.02	5.90	87964	171648
watt__1	24.5	6.81	3.72	278219	151964
watt__2	24.4	6.90	3.86	282451	158246
west0989	10.5	3.54	1.28	337264	121922
west2021	21.2	8.15	2.44	383530	115012

Table 4: (Part 2 of 2) The time it takes to compile generated code for each matrix using the `Unfolding` and `UnfoldingV2` methods.

compiler infrastructure's back-end to handle details of object file format. We directly write bits into a memory buffer to emit machine instructions.

2.1 Generating Machine-Level Code

We examined icc's output for the `UnfoldingV2` code to decide what machine instructions to emit. The statement

```
w[0] += vals[0] * v[2] + vals[1] * v[4] + vals[2] * v[6];
```

is translated by icc to X86_64 native code as follows:

```
movsd 16(%rdi), %xmm1        ;; xmm1 <- v[2]
movsd 32(%rdi), %xmm0        ;; xmm0 <- v[4]
movsd vals(%rip), %xmm4      ;; xmm4 <- vals[0]
movsd 8+vals(%rip), %xmm2    ;; xmm2 <- vals[1]
mulsd %xmm1, %xmm4           ;; xmm4 <- xmm4 * xmm1
mulsd %xmm0, %xmm2           ;; xmm2 <- xmm2 * xmm0
movsd 16+vals(%rip), %xmm3   ;; xmm3 <- vals[2]
addsd %xmm2, %xmm4           ;; xmm4 <- xmm4 + xmm2
movsd 48(%rdi), %xmm6        ;; xmm6 <- v[6]
;; omitted some instructions related to the next stmt
mulsd %xmm6, %xmm3           ;; xmm3 <- xmm3 * xmm6
addsd %xmm3, %xmm4           ;; xmm4 <- xmm4 + xmm3
addsd (%rsi), %xmm4          ;; xmm4 <- xmm4 + w[0]
;; omitted some instructions related to the other stmts
movsd %xmm4, (%rsi)          ;; w[0] <- xmm4
```

icc uses `movsd` instructions to load values into the `xmm` registers. It uses `addsd` and `mulsd` instructions to do both register-register and memory-register addition and multiplication operations. icc reorders the instructions to a great deal, but the general tendency we observed is that load operations are usually done in a batch to reduce memory latency. We also observed similar output from clang and gcc.

Inspired by icc's choice of instructions, for each row of the matrix, we emit code using the following strategy:

- Load as many elements of the vector `v` as possible into the `xmm` registers. E.g. for the statement given above, we do:

```
movsd 16(%rdi), %xmm0 ;; xmm0 <- v[2]
movsd 32(%rdi), %xmm1 ;; xmm1 <- v[4]
movsd 48(%rdi), %xmm2 ;; xmm2 <- v[6]
```

- Multiply matrix elements with the corresponding vector elements; keep the values in `xmm` registers. E.g. for the statement given above, we do:

```
mulsd (%rdx), %xmm0    ;; xmm0 <- xmm0 * vals[0]
mulsd 8(%rdx), %xmm1   ;; xmm1 <- xmm1 * vals[1]
mulsd 16(%rdx), %xmm2 ;; xmm2 <- xmm2 * vals[2]
```

- Add up the values stored in the `xmm` registers so that the final result is in `xmm0`. E.g. for the statement given above, we do:

```
addsd %xmm1, %xmm0     ;; xmm0 <- xmm0 + xmm1
addsd %xmm2, %xmm0     ;; xmm0 <- xmm0 + xmm2
```

  To reduce dependency, we perform add operations in a binary-tree fashion. For instance, if we are to reduce 10 `xmm` registers, we emit the following code:

```
addsd %xmm1, %xmm0     ;; xmm0 <- xmm0 + xmm1
addsd %xmm3, %xmm2     ;; xmm2 <- xmm2 + xmm3
addsd %xmm5, %xmm4     ;; xmm4 <- xmm4 + xmm5
addsd %xmm7, %xmm6     ;; xmm6 <- xmm6 + xmm7
```

```
addsd %xmm9, %xmm8      ;; xmm8 <- xmm8 + xmm9
addsd %xmm2, %xmm0      ;; xmm0 <- xmm0 + xmm2
addsd %xmm6, %xmm4      ;; xmm4 <- xmm4 + xmm6
addsd %xmm10, %xmm8     ;; xmm8 <- xmm8 + xmm10
addsd %xmm4, %xmm0      ;; xmm0 <- xmm0 + xmm4
addsd %xmm8, %xmm0      ;; xmm0 <- xmm0 + xmm8
```

- Load and add the output vector element onto the accumulated sum (which was in register `xmm0`), then write the result back to the output vector. E.g. for the statement given above, we do:

```
addsd (%rsi), %xmm0     ;; xmm0 <- xmm0 + w[0]
movsd %xmm0, (%rsi)     ;; w[0] <- xmm0
```

The number of `xmm` registers is 16. Therefore, the code generation strategy given above would run out of registers if a row has more than 16 nonzero elements. To remedy this problem, when there are more than 15 elements in a row, we calculate the result in chunks of 15 elements and accumulate results in the `xmm15` register.

We refer to the code generation algorithm discussed above as `MLUnfolding` (for "Machine-Level Unfolding"). A high-level overview of what `Unfolding`, `UnfoldingV2`, and `MLUnfolding` do is shown in Figure 5.

Conceptually, `MLUnfolding` is a straightforward mapping of the `UnfoldingV2` code into native code. A sophisticated instruction reordering algorithm, such as one that an industry-level compiler would apply, is not used here. Hence, a natural question that arises is how does `MLUnfolding` performs with respect to `UnfoldingV2`. Table 5 gives the comparison, where we report the ratio of the time taken by `UnfoldingV2`'s output to the time taken by `MLUnfolding`'s output. Having a value larger than 1 means that `MLUnfolding` gives better performance; these values are shown in the table in bold font. We see that `MLUnfolding` performs very well; on the average, its

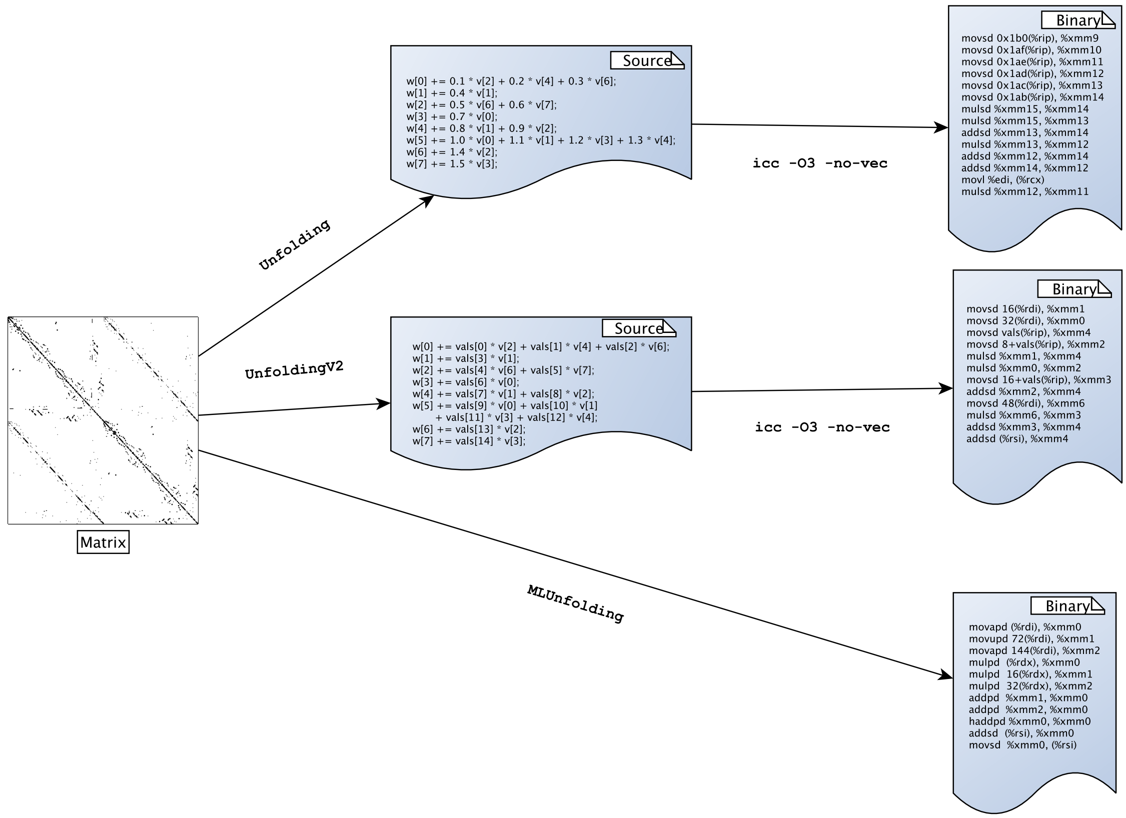

Figure 5: A high-level overview of the `Unfolding`, `UnfoldingV2`, and `MLUnfolding` methods.

performance is 1.09x the performance of `UnfoldingV2`. For 58 matrices out of 70, `MLUnfolding` gives better performance than `UnfoldingV2`.

Our motivation in generating native code ourselves instead of using a compiler was that the compiler takes too much time. So the next question to is, how much time does `MLUnfolding` take to generate code. The measured timings are given in Table 6. We see that code generation costs have dropped sharply, by roughly about three orders of magnitude.

In the following sections in this chapter, we discuss optimizations that we experimented with to improve the quality of `MLUnfolding`'s output.

Matrix	**MLUnfolding vs. UnfoldingV2**	**Matrix**	**MLUnfolding vs. UnfoldingV2**
dwt_419 (p)	**1.10**	tols4000	1.00
str__600	0.96	pores_2	**1.19**
minnesota (p)	**1.05**	spiral	**1.05**
bcspwr06 (p)	**1.06**	M80PI_n1	**1.09**
west0989	**1.03**	dw2048	**1.18**
bfw398a	**1.12**	watt__1	**1.12**
bcsstk19	1.00	watt__2	**1.04**
bcspwr08 (p)	**1.06**	bayer09	**1.11**
ck656	**1.02**	Pd	**1.04**
can__634 (p)	**1.06**	add20	**1.08**
tub1000	0.98	lshp3466 (p)	**1.10**
G33	**1.27**	dwt_2680 (p)	**1.04**
bcsstk06	1.00	as-735 (p)	**1.05**
hor__131	**1.07**	orsreg_1	**1.11**
gr_30_30	**1.08**	ca-GrQc (p)	**1.07**
pde900	**1.15**	adder_trans_02	**1.09**
cdde3	**1.31**	bcsstk26	**1.06**
bp__1600	**1.10**	plat1919	**1.05**
email (p)	**1.17**	wang2	**1.13**
steam2	**1.04**	coater1	**1.09**
gre_1107	**1.25**	add32	**1.06**
fs_760_1	**1.24**	olm5000	1.00
dwt_1242 (p)	**1.16**	rw5151	**1.19**
e05r0000	**1.02**	sherman5	0.97
fpga_dcop_51	0.99	saylr4	**1.14**
jpwh_991	**1.41**	Oregon-1 (p)	**1.07**
EVA (p)	**1.05**	mcfe	**1.11**
can_1072 (p)	**1.11**	lnsp3937	**1.04**
rdb1250	**1.08**	fidap002	**1.15**
west2021	**1.18**	bcsstk14	1.00
mahindas	0.98	cavity05	**1.05**
GD06_Java (p)	**1.22**	p2p-Gnutella04 (p)	**1.06**
nos3	0.99	mbeause	**1.23**
blckhole (p)	**1.05**	cry10000	**1.05**
c-18	0.94	mbeaflw	**1.24**

Table 5: The ratio of the running times of the codes generated using the `UnfoldingV2` approach (and compiled with icc) to the codes generated by the `MLUnfolding` approach. A value larger than 1 means `MLUnfolding` is faster.

Matrix	**PlainSpMV time (μs)**	**MLUnfolding code generation time (μs)**	**Ratio**	**Matrix**	**PlainSpMV time (μs)**	**MLUnfolding code generation time (μs)**	**Ratio**
add20	32.4	3959	122	GD06_Java (p)	23.2	2599	112
add32	62.6	6017	96	gr_30_30	8.0	1600	199
adder_trans_02	30.0	4071	136	gre_1107	14.5	1934	133
as-735 (p)	87.3	5115	59	hor__131	6.4	1431	225
bayer09	28.8	3881	135	jpwh_991	14.5	1986	137
bcspwr06 (p)	19.0	3232	170	lnsp3937	42.9	6805	159
bcspwr08 (p)	21.6	2815	130	lshp3466 (p)	34.8	4305	124
bcsstk06	6.4	1448	225	M80PI_n1	25.8	3663	142
bcsstk14	45.0	7988	177	mahindas	13.9	2418	174
bcsstk19	7.4	1492	201	mbeaflw	48.4	11315	234
bcsstk26	32.3	4442	138	mbeause	41.3	9452	229
bfw398a	5.9	1312	224	mcfe	26.4	5842	221
blckhole (p)	20.5	2809	137	minnesota (p)	24.5	3627	148
bp__1600	13.2	1670	127	nos3	11.9	2467	207
c-18	18.7	2842	152	olm5000	38.2	5948	156
ca-GrQc (p)	63.0	4656	74	Oregon-1 (p)	136.4	8291	61
can__634 (p)	7.7	3075	401	orsreg_1	24.3	3925	161
can_1072 (p)	15.5	2160	140	p2p-Gnutella04 (p)	134.7	10419	77
cavity05	36.7	7860	214	Pd	84.2	5341	63
cdde3	8.7	1689	193	pde900	8.1	1612	200
ck656	6.2	1411	227	plat1919	27.6	4750	172
coater1	27.2	5069	186	pores_2	14.5	2760	190
cry10000	88.7	14119	159	rdb1250	12.6	2299	183
dw2048	23.8	3114	131	rw5151	41.1	6030	147
dwt_1242 (p)	14.0	1971	141	saylr4	45.8	5966	130
dwt_2680 (p)	34.1	4168	122	sherman5	36.5	5879	161
dwt_419 (p)	5.8	1352	231	spiral	15.6	2953	189
e05r0000	6.4	1695	265	steam2	8.6	1766	206
email (p)	18.3	1876	103	str__600	6.1	1277	210
EVA (p)	37.3	2253	60	tols4000	23.3	3346	144
fidap002	26.1	6243	239	tub1000	7.4	1517	204
fpga_dcop_51	14.5	1972	136	wang2	34.3	5108	149
fs_760_1	9.2	1862	202	watt__1	24.5	3263	133
G33	13.0	3330	255	watt__2	24.4	3307	135
GD06_Java (p)	23.2	2599	112	west0989	10.5	1487	142
gr_30_30	8.0	1600	199	west2021	21.2	2619	123

Table 6: The time it takes to generate code for each matrix using the `MLUnfolding` method.

2.2 Optimization 1: Using Small Offsets When Accessing the Memory

Considering that unfolding results in long code, and that potentially induces negative impact on the instruction cache utilization, the first optimization we evaluate aims to decrease the generated code's size. We do this by using small offsets when accessing memory locations.

Matrix values in `MLUnfolding` are loaded from the memory in sequential order. Recall that we use a `mulsd` instruction to load a matrix element and immediately multiply it with an already-loaded vector element. In Figure 6 we show sample `mulsd` instructions and the corresponding byte values in hexadecimal format. In X86_64, instructions have variable lengths. Notice that when the offset is less than 128, a `mulsd` instruction takes 5 bytes. However, when the offset is 128 or more, the instruction takes 8 bytes. This means, for the first 16 elements[1] of the matrix, we will emit 5-byte instructions, but for each of the remaining thousands of elements, 3 extra bytes will be emitted.

In the `mulsd` instruction, `%rdx` is the register that holds the address of the starting point of the `vals` array. By shifting the value of `%rdx` forward in every 16 elements, we can always keep the offsets within the [0-120] range. We do this emitting a `leaq` (load effective address) instruction just before the offset is about to become 128. A sample

[1]Double precision values are stored in 8 bytes; therefore, the offset grows in increments of 8. We reach 128 from 0 in 16 steps.

```
mulsd 104(%rdx), %xmm0    |    f2 0f 59 42 68
mulsd 112(%rdx), %xmm0    |    f2 0f 59 42 70
mulsd 120(%rdx), %xmm0    |    f2 0f 59 42 78
mulsd 128(%rdx), %xmm0    |    f2 0f 59 82 80 00 00 00
mulsd 136(%rdx), %xmm0    |    f2 0f 59 82 88 00 00 00
mulsd 144(%rdx), %xmm0    |    f2 0f 59 82 90 00 00 00
```

Figure 6: Sample `mulsd` instructions with offset values around 128, and the corresponding X86_64 instructions in hexadecimal format.

```
mulsd 104(%rdx), %xmm0
mulsd 112(%rdx), %xmm1

mulsd 120(%rdx), %xmm2
mulsd 128(%rdx), %xmm3
mulsd 136(%rdx), %xmm4
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm2, %xmm0
addsd %xmm4, %xmm0
addsd %xmm0, %xmm7
addsd 8(%rsi), %xmm7
movsd %xmm7, 8(%rsi)
xorps %xmm7, %xmm7
movsd 8(%rdi), %xmm0
movsd 16(%rdi), %xmm1
movsd 24(%rdi), %xmm2
movsd 80(%rdi), %xmm3
movsd 88(%rdi), %xmm4
movsd 96(%rdi), %xmm5
movsd 152(%rdi), %xmm6
mulsd 144(%rdx), %xmm0
mulsd 152(%rdx), %xmm1
mulsd 160(%rdx), %xmm2
mulsd 168(%rdx), %xmm3
mulsd 176(%rdx), %xmm4
mulsd 184(%rdx), %xmm5
mulsd 192(%rdx), %xmm6
```

```
mulsd 104(%rdx), %xmm0
mulsd 112(%rdx), %xmm1
leaq  120(%rdx), %rdx
mulsd (%rdx), %xmm2      ;; offset reduced
mulsd 8(%rdx), %xmm3     ;; offset reduced
mulsd 16(%rdx), %xmm4    ;; offset reduced
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm2, %xmm0
addsd %xmm4, %xmm0
addsd %xmm0, %xmm7
addsd 8(%rsi), %xmm7
movsd %xmm7, 8(%rsi)
xorps %xmm7, %xmm7
movsd 8(%rdi), %xmm0
movsd 16(%rdi), %xmm1
movsd 24(%rdi), %xmm2
movsd 80(%rdi), %xmm3
movsd 88(%rdi), %xmm4
movsd 96(%rdi), %xmm5
movsd 152(%rdi), %xmm6
mulsd 24(%rdx), %xmm0    ;; offset reduced
mulsd 32(%rdx), %xmm1    ;; offset reduced
mulsd 40(%rdx), %xmm2    ;; offset reduced
mulsd 48(%rdx), %xmm3    ;; offset reduced
mulsd 56(%rdx), %xmm4    ;; offset reduced
mulsd 64(%rdx), %xmm5    ;; offset reduced
mulsd 72(%rdx), %xmm6    ;; offset reduced
```

Figure 7: `MLUnfolding` produces the code on the left. Applying the offset-reducing optimization gives the code on the right.

before/after comparison is given in Figure 7. Here, the code emitted by `MLUnfolding` is given on the left-hand-side. On the right-hand-side, we see the code after emitting a `leaq` to increase the value of `%rdx` by 120. We do not attempt to reduce the offsets of the `%rdi` register in the `movsd` instructions because they are used for loading vector `v`'s elements. Unlike the accesses to `vals`, accesses to `v` are arbitrary and hence the offsets here do not necessarily increase monotonically. However, the accesses to the output vector `w` are consecutive. Hence, we apply the same offset-reducing optimization to `w` as well. This is not shown in Figure 7 due to space concerns.

A `leaq` instruction that increments the value of `%rdx` by 120 consumes 4 bytes. However, it saves us 3 bytes 15 times. Therefore, for every 15 elements of the matrix, we gain 45 bytes by compromising 4 bytes.

```
movsd (%rdi), %xmm0        f2 0f 10 07
movsd (%rdi), %xmm1        f2 0f 10 0f
movsd (%rdi), %xmm2        f2 0f 10 17
movsd (%rdi), %xmm3        f2 0f 10 1f
movsd (%rdi), %xmm4        f2 0f 10 27
movsd (%rdi), %xmm5        f2 0f 10 2f
movsd (%rdi), %xmm6        f2 0f 10 37
movsd (%rdi), %xmm7        f2 0f 10 3f
movsd (%rdi), %xmm8        f2 44 0f 10 07
movsd (%rdi), %xmm9        f2 44 0f 10 0f
movsd (%rdi), %xmm10       f2 44 0f 10 17
movsd (%rdi), %xmm11       f2 44 0f 10 1f
movsd (%rdi), %xmm12       f2 44 0f 10 27
movsd (%rdi), %xmm13       f2 44 0f 10 2f
movsd (%rdi), %xmm14       f2 44 0f 10 37
movsd (%rdi), %xmm15       f2 44 0f 10 3f
```

Figure 8: Sample X86_64 instructions that use `xmm` registers and their corresponding hexadecimal format. Using an `xmm` register with a number 8 or more consumes an extra byte.

In Table 7, we report the impact of the offset-reducing optimization with respect to `MLUnfolding`. We show the performance and the size of the code, both relative to the performance and size of code produced by `MLUnfolding`, respectively. For almost all cases performance is increased and code size is decreased significantly. On the average, the performance is 1.19x, and the code size is 0.83x of `MLUnfolding`.

2.3 Optimization 2: Using a Restricted Set of Registers

Another optimization we experimented with again aims to reduce the code size. Let us first take a look at how `xmm` registers effect the instruction lengths. In Figure 8, we show sample instructions in ASCII and hexadecimal format.

Recall from `MLUnfolding` that the multiplication of a matrix value and a vector element is stored in an `xmm` register. Using a `xmm` register numbered 8-15 consumes an extra byte in the instruction as opposed to using a register with number 0-7. To save space, we limit the set of the available registers to `xmm0-xmm7` and we do not use `xmm8-xmm15`.

Limiting the set of `xmm` registers to 0-7 will have no effect on rows that have fewer

Matrix	Performance wrt MLUnfolding	Object code size wrt MLUnfolding	Matrix	Performance wrt MLUnfolding	Object code size wrt MLUnfolding
dwt_419 (p)	1.45	0.83	tols4000	1.31	0.79
str__600	1.19	0.85	pores_2	1.13	0.84
minnesota (p)	1.28	0.76	spiral	1.35	0.84
bcspwr06 (p)	1.25	0.79	M80PI_n1	1.29	0.79
west0989	1.23	0.81	dw2048	1.34	0.82
bfw398a	1.18	0.85	watt__1	1.17	0.83
bcsstk19	1.02	0.82	watt__2	1.33	0.83
bcspwr08 (p)	1.24	0.79	bayer09	1.20	0.82
ck656	1.20	0.83	Pd	1.22	0.77
can__634 (p)	1.18	0.84	add20	1.17	0.83
tub1000	1.21	0.82	lshp3466 (p)	1.22	0.81
G33	1.16	0.78	dwt_2680 (p)	1.21	0.82
bcsstk06	1.17	0.85	as-735 (p)	1.19	0.79
hor__131	1.18	0.85	orsreg_1	1.19	0.83
gr_30_30	1.21	0.82	ca-GrQc (p)	1.27	0.82
pde900	1.21	0.82	adder_trans_02	1.15	0.84
cdde3	1.21	0.82	bcsstk26	1.22	0.84
bp__1600	1.19	0.83	plat1919	1.17	0.85
email (p)	0.86	0.83	wang2	1.17	0.83
steam2	1.18	0.85	coater1	1.23	0.86
gre_1107	1.20	0.83	add32	1.20	0.82
fs_760_1	0.99	0.84	olm5000	1.23	0.82
dwt_1242 (p)	1.02	0.82	rw5151	1.19	0.81
e05r0000	1.15	0.86	sherman5	1.16	0.84
fpga_dcop_51	1.17	0.83	saylr4	1.17	0.83
jpwh_991	1.20	0.83	Oregon-1 (p)	1.21	0.79
EVA (p)	1.14	0.84	mcfe	1.16	0.86
can_1072 (p)	1.10	0.83	lnsp3937	1.19	0.83
rdb1250	1.36	0.83	fidap002	1.17	0.87
west2021	1.08	0.81	bcsstk14	1.17	0.86
mahindas	1.21	0.84	cavity05	1.17	0.86
GD06_Java (p)	1.12	0.83	p2p-Gnutella04 (p)	1.13	0.84
nos3	1.27	0.84	mbeause	1.12	0.87
blckhole (p)	1.37	0.82	cry10000	1.18	0.82
c-18	1.16	0.82	mbeaflw	1.14	0.87

Table 7: The impact of offset-reducing optimization on performance and object code size with respect to `MLUnfolding`.

```
;; Row has 12 nonzero elements
movsd (%rdi), %xmm0
movsd 8(%rdi), %xmm1
movsd 16(%rdi), %xmm2
movsd 24(%rdi), %xmm3
movsd 72(%rdi), %xmm4
movsd 80(%rdi), %xmm5
movsd 88(%rdi), %xmm6
movsd 96(%rdi), %xmm7
movsd 144(%rdi), %xmm8
movsd 152(%rdi), %xmm9
movsd 160(%rdi), %xmm10
movsd 168(%rdi), %xmm11
mulsd 48(%rdx), %xmm0
mulsd 56(%rdx), %xmm1
mulsd 64(%rdx), %xmm2
mulsd 72(%rdx), %xmm3
mulsd 80(%rdx), %xmm4
mulsd 88(%rdx), %xmm5
mulsd 96(%rdx), %xmm6
mulsd 104(%rdx), %xmm7
mulsd 112(%rdx), %xmm8
mulsd 120(%rdx), %xmm9
mulsd 128(%rdx), %xmm10
mulsd 136(%rdx), %xmm11
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm5, %xmm4
addsd %xmm7, %xmm6
addsd %xmm9, %xmm8
addsd %xmm11, %xmm10
addsd %xmm2, %xmm0
addsd %xmm6, %xmm4
addsd %xmm10, %xmm8
addsd %xmm4, %xmm0
addsd %xmm8, %xmm0
```

```
;; Row has 12 nonzero elements
movsd (%rdi), %xmm0
movsd 8(%rdi), %xmm1
movsd 16(%rdi), %xmm2
movsd 24(%rdi), %xmm3
movsd 72(%rdi), %xmm4
movsd 80(%rdi), %xmm5
movsd 88(%rdi), %xmm6 ;; reached reg. limit
mulsd 48(%rdx), %xmm0
mulsd 56(%rdx), %xmm1
mulsd 64(%rdx), %xmm2
mulsd 72(%rdx), %xmm3
mulsd 80(%rdx), %xmm4
mulsd 88(%rdx), %xmm5
mulsd 96(%rdx), %xmm6
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm5, %xmm4
addsd %xmm2, %xmm0
addsd %xmm6, %xmm4
addsd %xmm4, %xmm0
addsd %xmm0, %xmm7 ;; save partial result
movsd 96(%rdi), %xmm0
movsd 144(%rdi), %xmm1
movsd 152(%rdi), %xmm2
movsd 160(%rdi), %xmm3
movsd 168(%rdi), %xmm4
mulsd 104(%rdx), %xmm0
mulsd 112(%rdx), %xmm1
mulsd 120(%rdx), %xmm2
mulsd 128(%rdx), %xmm3
mulsd 136(%rdx), %xmm4
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm2, %xmm0
addsd %xmm4, %xmm0
addsd %xmm0, %xmm7
```

Figure 9: Left: code generated by `MLUnfolding` method. Right: code obtained when the `xmm` register set is limited to 0-7.

than 8 elements. However, if there are more elements, the partial result will have to be stored in the accumulator register more often than using `xmm0-xmm15`. This means, while we are saving space by using small register numbers, we will have to emit extra instructions (and thus lose space) to accumulate partial results.

Figure 9 illustrates the impact of this optimization when there are 12 nonzero elements in a row. Here, the left-hand-side shows the code generated by the `MLUnfolding` method; the right-hand-side shows the same code when the register set is limited to `xmm0-xmm7`. On the left, 12 elements of the vector are loaded into `xmm` registers 0-11 at once. Then, 12 multiplication operations are performed. Finally, the results of multiplications are added together into `xmm0`. On the right, because the registers are limited to 0-7, the first 7 registers are used to load vector elements. Then, 7 multiplication operations are performed, followed by addition operations that calculate the sum of 7 multiplications into `xmm0`. This partial result is then put into the accumulator register `xmm7`. The right-hand-side has an extra register-register `addsd` instruction. The length of this instruction is 4 bytes. The left-hand-side has 4 `movsd`, 4 `mulsd`, and 4 `addsd` instructions that use an `xmm` register in range 8-15; giving, in total 12 extra bytes. So, in this example 12 bytes were saved while 4 bytes were introduced for an additional register-register instruction.

Table 8 lists the performance, object code size, and the instruction count when register set limiting optimization is applied on top of `MLUnfolding`. All the numbers are relative values with respect to the corresponding values of `MLUnfolding`. We see that the performance of the code may change both positively and negatively: In the worst case, dwt_1242's performance went down to 0.74x; in the best case, 1.12x performance was obtained for rdb1250. Changes in the code size are less variable; the minimum size is 0.93x. On the average, this optimization resulted in 0.98x performance while reducing the code size to only 0.99x. Better results may have been achieved if this optimization was turned on/off for each row depending on the number

Matrix	Performance wrt MLUnfolding	Code size wrt MLUnfolding	Instruction count wrt MLUnfolding	Matrix	Performance wrt MLUnfolding	Code size wrt MLUnfolding	Instruction count wrt MLUnfolding
dwt_419 (p)	0.99	1.00	1.02	tols4000	0.98	0.99	1.00
str__600	1.06	0.95	1.01	pores_2	0.91	1.01	1.04
minnesota (p)	1.00	1.00	1.00	spiral	1.05	0.95	1.00
bcspwr06 (p)	1.00	1.00	1.00	M80PI_n1	0.96	1.00	1.00
west0989	1.00	1.01	1.02	dw2048	0.93	1.00	1.00
bfw398a	1.01	0.99	1.03	watt__1	0.97	1.00	1.00
bcsstk19	0.97	1.00	1.00	watt__2	1.06	1.00	1.00
bcspwr08 (p)	0.86	1.00	1.00	bayer09	0.97	0.99	1.02
ck656	0.85	1.00	1.04	Pd	1.00	1.00	1.00
can__634 (p)	1.00	0.99	1.03	add20	1.03	0.98	1.00
tub1000	1.00	1.00	1.00	lshp3466 (p)	0.96	1.00	1.00
G33	0.98	1.00	1.00	dwt_2680 (p)	1.00	1.01	1.02
bcsstk06	1.04	0.98	1.04	as-735 (p)	1.02	1.00	1.00
hor__131	1.01	1.00	1.05	orsreg_1	1.01	1.00	1.00
gr_30_30	1.00	1.00	1.00	ca-GrQc (p)	1.05	0.99	1.01
pde900	0.87	1.00	1.00	adder_trans_02	1.00	1.00	1.03
cdde3	0.77	1.00	1.00	bcsstk26	0.98	0.98	1.03
bp__1600	1.01	0.99	1.02	plat1919	1.03	0.98	1.05
email (p)	1.00	0.99	1.02	wang2	1.00	1.00	1.00
steam2	0.84	0.98	1.05	coater1	1.07	0.96	1.02
gre_1107	0.89	1.00	1.00	add32	1.00	1.00	1.01
fs_760_1	0.96	0.99	1.03	olm5000	1.00	1.00	1.00
dwt_1242 (p)	0.74	1.00	1.01	rw5151	1.00	1.00	1.00
e05r0000	1.10	0.94	1.01	sherman5	1.04	0.97	1.02
fpga_dcop_51	0.80	0.99	1.00	saylr4	0.99	1.00	1.00
jpwh_991	0.79	1.01	1.03	Oregon-1 (p)	1.00	0.99	1.00
EVA (p)	1.04	0.96	1.00	mcfe	1.09	0.93	1.00
can_1072 (p)	0.75	1.01	1.03	lnsp3937	0.99	1.00	1.03
rdb1250	1.12	1.00	1.00	fidap002	1.07	0.93	1.00
west2021	1.04	1.01	1.02	bcsstk14	1.05	0.95	1.01
mahindas	1.09	0.97	1.01	cavity05	1.08	0.94	1.00
GD06_Java (p)	0.89	0.98	1.02	p2p-Gnutella04 (p)	1.00	1.00	1.06
nos3	1.00	1.01	1.06	mbeause	1.05	0.93	1.00
blckhole (p)	0.95	1.00	1.00	cry10000	1.01	1.00	1.00
c-18	0.84	0.99	1.02	mbeaflw	1.06	0.93	1.00

Table 8: The speedup, code size reduction, and instruction count increases imposed by the register set limiting optimization with respect to `MLUnfolding`.

of nonzero elements of the row. However, we have not tested this idea. We tested this optimization also on another matrix set that includes larger matrices. There, we saw the optimization to provide around 3% speedup relatively consistently. Thus, we decided to include this optimization in the final version of our code generator.

2.4 Optimization 3: Using a Pool of Distinct Values

In Section 1.3.1 we commented on how having few distinct values can enable significant performance improvements. As another optimization attempt, we perform a distinct value analysis to emit more efficient code. In this analysis, we identify distinct values of a matrix, and put these values in a pool. For each row, multiplications that have the same constant multiplier are grouped together and the reverse of distribution of multiplication over addition is applied; i.e. $c \times a + c \times b = c \times (a + b)$. Hence, for each group, we emit addition operations followed by a single multiplication operation. In this section we show code snippets to explain how this optimization is realized.

Let us suppose that we are to emit assembly code for the following computation, represented in source code:

```
w[2] += 7*v[106] + 7*v[329] + 7*v[1040] + 7*v[4952] + 3*v[19247];
w[3] += 7*v[129] + 7*v[201] + 7*v[329] + 7*v[14911];
```

After grouping the vector elements for the same constant value, we essentially emit code corresponding to the following computation:

```
w[2] += 7*(v[106] + v[329] + v[1040] + v[4952]) + 3*v[19247];
w[3] += 7*(v[129] + v[201] + v[329] + v[14911]);
```

```
movsd 848(%rdi), %xmm0      ; xmm0 <- v[106]
movsd 2632(%rdi), %xmm1     ; xmm1 <- v[329]
addsd 8320(%rdi), %xmm0     ; xmm0 <- v[106] + v[1040]
addsd 39616(%rdi), %xmm1    ; xmm1 <- v[329] + v[4952]
addsd %xmm1, %xmm0          ; xmm0 <- v[106] + v[329] + v[1040] + v[4952]
mulsd (%rdx), %xmm0         ; xmm0 <- vals[0] * (v[106] + v[329] + v[1040] + v[4952])
addsd %xmm0, %xmm15         ; save the partial result
movsd 153976(%rdi), %xmm0   ; xmm0 <- v[19247]
mulsd 8(%rdx), %xmm0        ; xmm0 <- vals[1] * v[19247]
addsd %xmm0, %xmm15         ; xmm15 <- vals[0] * (v[106] + v[329] + v[1040] + v[4952])
                            ;         + vals[1] * v[19247]
addsd 16(%rsi), %xmm15      ; xmm15 <- xmm15 + w[2]
movsd %xmm15, 16(%rsi)      ; w[2] <- xmm15
movsd 1032(%rdi), %xmm0     ; xmm0 <- v[129]
movsd 1608(%rdi), %xmm1     ; xmm1 <- v[201]
addsd 2632(%rdi), %xmm0     ; xmm0 <- v[129] + v[329]
addsd 119288(%rdi), %xmm1   ; xmm1 <- v[129] + v[14911]
addsd %xmm1, %xmm0          ; xmm0 <- v[129] + v[201] + v[329] + v[14911]
mulsd (%rdx), %xmm0         ; xmm0 <- vals[0] * (v[129] + v[201] + v[329] + v[14911])
addsd 24(%rsi), %xmm0       ; xmm0 <- xmm0 + w[3]
movsd %xmm0, 24(%rsi)       ; w[3] <- xmm0
```

Figure 10: A sample assembly code that performs computation according to distinct values.

Recall that the floating point constants are emitted to the data section as a pool. So, what we have is

```
double vals[] = {7, 3};
w[2] += vals[0]*(v[106] + v[329] + v[1040] + v[4952]) + vals[1]*v[19247];
w[3] += vals[0]*(v[129] + v[201] + v[329] + v[14911]);
```

Figure 10 shows the assembly code we generate that corresponds to this example.

Note that the distinct value optimization requires that we have access to the matrix values. The previous two optimizations, namely, the offset-reducing optimization and the register set limiting optimization, did not make any use of actual matrix values.

As part of this optimization we also apply two arithmetic optimizations:

- $... + 1 \times a = ... + a$. In this case we simply omit the multiplication.

- $... + -1 \times a = ... - a$. In this case we omit the multiplication and emit a subtraction instruction instead of addition.

Distinct value optimization is expected to bring substantial performance improvements when there are few distinct values because it

- reduces the number of matrix elements loaded from the memory,
- reduces the number of floating point operations,
- reduces the code size by eliminating instructions.

However, this optimization also has drawbacks when the distinct values are not few:

- Because the values are put into a pool, matrix elements are not necessarily accessed sequentially. For this reason, offset-reducing optimization cannot be applied to the `vals` array.
- Not accessing the elements sequentially may have negative impact on cache utilization.
- If the number of distinct values is close to the number of nonzero elements (i.e. very few common elements in the matrix), the emitted code will be very long due to premature partial result accumulations.

Table 9 shows the speedup and code size reduction obtained by this optimization with respect to `MLUnfolding`. In this table, we sort the matrices in ascending order according to the percentage of their distinct values. So the matrices close to the top have fewer distinct values. It is clear from this table that distinct value optimization brings substantial improvement for matrices with relatively few number of distinct values. (One exception is gre_1107, which has only 11 distinct values, but we got 0.94x slowdown in performance.) The optimization has negative impact on matrices that do not have few distinct values.

Matrix	**Distinct values (#)**	**Distinct values (%)**	**Performance wrt MLUnfolding**	**Code size wrt MLUnfolding**	**Matrix**	**Distinct values (#)**	**Distinct values (%)**	**Performance wrt MLUnfolding**	**Code size wrt MLUnfolding**
olm5000	6	0%	1.20	0.78	west0989	1776	50%	0.99	0.88
gr_30_30	2	0%	1.39	0.63	c-18	4861	56%	1.01	0.88
saylr4	11	0%	1.40	0.68	add20	7390	56%	1.07	0.91
G33	2	0%	1.30	0.63	pores_2	5407	56%	0.96	0.94
rdb1250	6	0%	1.52	0.70	watt__2	6589	57%	1.03	1.00
cdde3	5	0%	1.03	0.87	watt__1	6524	57%	0.98	1.00
gre_1107	11	0%	0.94	0.84	west2021	4235	58%	0.94	0.92
jpwh_991	14	0%	1.49	0.60	str__600	1972	60%	1.05	0.88
M80PI_n1	70	1%	1.27	0.76	add32	13883	70%	0.94	0.97
rw5151	150	1%	1.29	0.74	adder_trans_02	10327	71%	0.96	0.95
orsreg_1	111	1%	1.21	0.85	sherman5	15096	73%	1.00	0.91
nos3	149	2%	1.28	0.90	pde900	3248	74%	0.89	1.01
bfw398a	92	3%	1.22	0.75	ck656	3054	79%	0.91	1.00
Pd	432	3%	1.30	0.66	fs_760_1	4743	83%	0.93	0.98
mbeause	2100	5%	1.59	0.65	bcsstk26	13480	84%	0.96	0.97
dw2048	693	7%	0.97	0.94	plat1919	17120	100%	0.95	0.96
coater1	1380	7%	1.37	0.78	cry10000	49599	100%	0.90	1.01
wang2	1727	9%	1.08	0.90	mcfe	24381	100%	1.02	0.93
cavity05	3280	10%	1.16	0.88	as-735 (p)	13895	100%	0.91	0.96
fpga_dcop_51	953	16%	1.06	0.90	bcspwr06 (p)	3377	100%	0.87	0.99
lnsp3937	4176	16%	1.01	0.95	bcspwr08 (p)	3837	100%	0.87	0.99
steam2	1071	19%	1.00	0.93	blckhole (p)	8502	100%	0.83	1.01
e05r0000	1269	22%	0.88	0.90	ca-GrQc (p)	14496	100%	0.94	0.96
bcsstk06	1045	25%	0.97	0.96	can__634 (p)	3931	100%	0.91	0.99
spiral	3089	31%	1.30	0.85	can_1072 (p)	6758	100%	0.86	1.01
tols4000	3188	36%	0.98	0.87	dwt_1242 (p)	5834	100%	0.80	1.01
hor__131	1553	37%	1.03	0.90	dwt_2680 (p)	13853	100%	0.89	1.01
bp__1600	1803	37%	1.13	0.80	dwt_419 (p)	1991	100%	0.88	1.01
mbeaflw	19778	40%	0.98	0.93	email (p)	5451	100%	0.62	0.98
fidap002	11118	41%	1.07	0.91	EVA (p)	6726	100%	0.89	0.94
bayer09	5003	43%	0.98	0.90	GD06_Java (p)	8032	100%	0.83	0.97
mahindas	3291	43%	1.05	0.87	lshp3466 (p)	13681	100%	0.84	1.01
bcsstk14	14044	43%	1.02	0.93	minnesota (p)	3303	100%	0.91	0.91
bcsstk19	1852	48%	0.90	1.00	Oregon-1 (p)	23409	100%	0.89	0.96
tub1000	1990	50%	0.94	0.95	p2p-Gnutella04 (p)	39994	100%	0.96	0.98

Table 9: The speedup and code size reduction obtained by distinct value optimization with respect to `MLUnfolding`. Matrices are sorted according to the percentage of their distinct values.

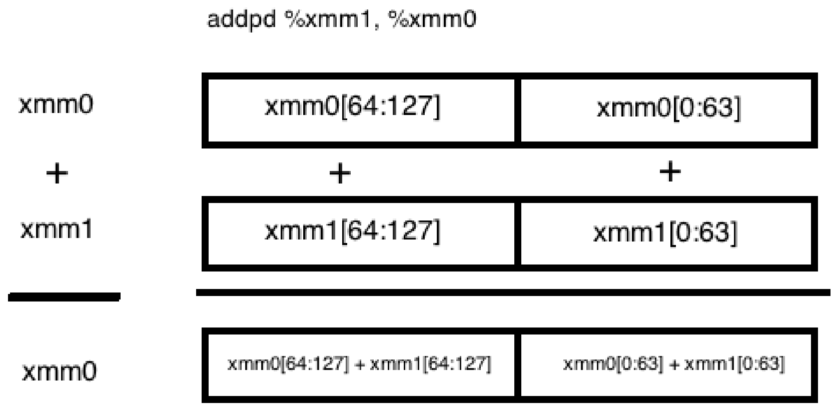

Figure 11: ADDPD instruction

2.5 Optimization 4: Using Vector Instructions

Out target architecture X86_64 has SIMD (Single Instruction Multiple Data) instructions that operate on vector registers (i.e. `xmm`'s). The `xmm` registers are 128-bit; they can hold two double-precision floating point values, one in the upper 64 bits and the other in the lower 64 bits. Vector instructions allow performing calculations with the two halves of `xmm` registers simultaneously. An example is the `addpd` instruction. It adds the lower and upper halves of its operands independently at the same time, then writes the results to the lower and upper halves of its destination. This is illustrated in Figure 11.

In the code generated by `MLUnfolding`, matrix values are accessed consecutively. Depending on the shape of the matrix, there may be cases where the vector elements, with which matrix elements are multiplied, are also accessed consecutively. For instance, after unfolding, we may see a case such as

```
... + vals[4] * v[18] + vals[5] * v[19] + ...
```

For the multiplication operations in this code, `MLUnfolding` produces the following assembly:

```
movsd 144(%rdi), %xmm4 ; xmm4 <- v[18]
movsd 152(%rdi), %xmm5 ; xmm5 <- v[19]
...
mulsd 32(%rdx), %xmm4   ; xmm4 <- xmm4 + vals[4]
mulsd 40(%rdx), %xmm5   ; xmm5 <- xmm5 + vals[5]
```

Here, two consecutive elements from the `vals` array, and two consecutive elements from the `v` array are loaded. Vectorized computation is ideal for this situation. So, the computation above can be done as shown below, where we use '::' to denote concatenation of the lower and upper parts of a vector register.

```
movapd 144(%rdi), %xmm4   ; xmm4 <- v[18] :: v[19]
...
mulpd  32(%rdx), %xmm4    ; xmm4 <- vals[4] * v[18] :: vals[5] * v[19]
```

Above, we show uses of the `movapd` and `mulpd` instructions. The `movapd` instruction moves 128 bits from the given memory location to the destination register. It requires that the memory location is aligned to 128 bits. When the memory is not aligned, we make use of the `movupd` instruction. The `mulpd` instruction multiplies two consecutive 64-bit values read from the memory with lower and upper halves of a vector register, and then writes the results to the corresponding halves of the register. (There is also a register-register version of the `mulpd` instruction that works exactly the same way as `addpd` as explained in Figure 11.) `mulpd` requires the memory location to be aligned to 128 bits. Finally, we also use the `haddpd` instruction that performs "horizontal add" on a vector instruction: sums up the lower and upper halves of its source register, write the result into the lower half of its destination register. A bigger example that illustrates the use of these instructions is given in Figure 12.

We implemented a vectorized version of unfolding. In this version we analyze the matrix data and identify pairs of elements (e.g. elements that form an expression like

```
w[0] += vals[0] * v[0] + vals[1] * v[1]
      + vals[2] * v[9] + vals[3] * v[10]
      + vals[4] * v[18] + vals[5] * v[19];

movapd (%rdi), %xmm0       ; xmm0 <- v[0] :: v[1]
movupd 72(%rdi), %xmm1     ; xmm1 <- v[9] :: v[10]
movapd 144(%rdi), %xmm2    ; xmm2 <- v[18] :: v[19]
mulpd  (%rdx), %xmm0       ; xmm0 <- vals[0] * v[0] :: vals[1] * v[1]
mulpd  16(%rdx), %xmm1     ; xmm1 <- vals[2] * v[9] :: vals[3] * v[10]
mulpd  32(%rdx), %xmm2     ; xmm2 <- vals[4] * v[18] :: vals[5] * v[19]
addpd  %xmm1, %xmm0        ; xmm0[0:63]    <- xmm0[0:63]    + xmm1[0:63]
                           ; xmm0[64:128] <- xmm0[64:128] + xmm1[64:128]
addpd  %xmm2, %xmm0        ; xmm0[0:63]    <- xmm0[0:63]    + xmm2[0:63]
                           ; xmm0[64:128] <- xmm0[64:128] + xmm2[64:128]
haddpd %xmm0, %xmm0        ; xmm0[0:63]    <- xmm0[0:63] + xmm0[64:128]
addsd  (%rsi), %xmm0       ; xmm0 <- xmm0 + w[0]
movsd  %xmm0, (%rsi)       ; w[0] <- xmm0
```

Figure 12: A sample statement in C and its vectorized code in assembly.

`vals`$[i] \times$ `v`$[p] +$ `vals`$[j] \times$ `v`$[q]$) that are vectorizable. For a pair to be vectorizable, the following conditions must hold:

- The accessed matrix elements must be consecutive, i.e. $j = i + 1$.
- The accessed vector elements must be consecutive, i.e. $q = p + 1$.
- The accessed matrix elements must be aligned to 128-bits, i.e. i must be a multiple of 128/8. This condition is required to be able to do `mulpd`.

In `MLUnfolding`, all the accesses to the matrix elements are consecutive. So the first condition above is easily satisfied. Table 10 gives the number of vectorized pairs and the performance obtained when vectorization is applied on top of `MLUnfolding`. On the average, 1.19x performance is achieved.

The offset-reducing and register set restriction optimizations are orthogonal to vectorizability; they can be applied together in harmony. Distinct value optimization, however, does not work in favor of vectorization. Because the distinct value analysis creates a pool of values, the emitted code does not contain consecutive accesses to matrix values anymore. This makes the code less prone to vectorization. In our

Matrix	NZ	Vectorized pairs	Performance wrt MLUnfolding	Matrix	NZ	Vectorized pairs	Performance wrt MLUnfolding
dwt_419	1991	456	1.44	tols4000	8784	756	1.04
str__600	3279	642	1.26	pores_2	9613	1507	1.08
minnesota	3303	50	1.00	spiral	9831	3719	1.64
bcspwr06	3377	257	1.05	M80PI_n1	9927	1833	1.16
west0989	3518	418	1.10	dw2048	10114	1966	1.06
bfw398a	3678	120	1.02	watt__1	11360	1674	1.11
bcsstk19	3835	1098	1.28	watt__2	11550	1735	1.15
bcspwr08	3837	278	1.04	bayer09	11767	1443	1.11
ck656	3884	1042	1.31	Pd	13036	1062	1.04
can__634	3931	1045	1.11	add20	13151	1743	1.11
tub1000	3996	998	1.20	lshp3466	13681	3302	1.19
G33	4000	78	1.01	dwt_2680	13853	3315	1.27
bcsstk06	4140	1399	1.51	as-735	13895	76	1.00
hor__131	4182	530	1.12	orsreg_1	14133	2100	1.19
gr_30_30	4322	1262	1.28	ca-GrQc	14496	79	1.05
pde900	4380	842	1.16	adder_trans_02	14579	1630	1.08
cdde3	4681	930	1.16	bcsstk26	16129	4164	1.31
bp__1600	4841	236	1.04	plat1919	17159	4589	1.32
email	5451	203	1.03	wang2	19093	2829	1.10
steam2	5660	828	1.14	coater1	19457	2776	1.14
gre_1107	5664	381	0.80	add32	19848	3021	1.11
fs_760_1	5739	425	1.04	olm5000	19996	9998	1.50
dwt_1242	5834	1528	1.23	rw5151	20199	2	1.01
e05r0000	5846	2500	1.73	sherman5	20793	6218	1.33
fpga_dcop_51	5892	774	1.14	saylr4	22316	3040	1.08
jpwh_991	6027	92	1.02	Oregon-1	23409	135	1.00
EVA	6726	2013	1.30	mcfe	24382	6974	1.37
can_1072	6758	1009	1.12	lnsp3937	25407	1364	1.04
rdb1250	7300	1248	1.11	fidap002	26831	11832	1.87
west2021	7310	893	1.04	bcsstk14	32630	12136	1.55
mahindas	7682	1575	1.25	cavity05	32632	11916	1.56
GD06_Java	8032	673	0.94	p2p-Gnutella04	39994	3809	1.06
nos3	8402	3188	1.66	mbeause	41063	14251	1.49
blckhole	8502	1926	1.03	cry10000	49699	9803	1.15
c-18	8657	912	1.07	mbeaflw	49920	16781	1.45

Table 10: The number of vectorized pairs and performance with respect to `MLUnfolding` when vectorization optimization is applied to `MLUnfolding`.

experiments we have seen seldom improvement in performance when vectorization is applied on top of distinct value analysis, therefore we exclude their combination from our tests and reports.

2.6 Optimization 5: Embedding Matrix Values into the Text Section of the Code

In this section we discuss a transformation that caused slowdown in the performance. We still dedicate a section to this approach so that ideas that did not work are also documented for future reference.

Recall that in the native code produced by the compiler, matrix values that appear as constants in the source code are emitted to the data section of the object code. These values are loaded from the memory as if they were values in an array. Instead of emitting matrix values in the data section, we experimented with an approach where the values are moved into registers from immediate values. Hence, the values are embedded directly in the text section of the code. A before/after comparison is shown in Figure 13. Here, the left-hand-side shows the original assembly emitted by the `MLUnfolding` method; the right-hand-side shows the assembly when the matrix values are embedded in the instructions are immediate values. This approach increases the code size significantly. However, our motivation in experimenting with this code was that at the L1 level, CPU has separate caches for instruction and data. To maximize the utilization, moving part of the data to the instruction side is a feasible idea.

When we measured the performance, we saw that embedding the data in the instructions causes significant slowdown for all of the matrices in our set. On the average, there is 27% slowdown. The smallest code size is 60KB (for dwt_419, which has 1991 nonzero values). The machine on which we run experiments has 32K L1 instruction and data caches. Because the code size for the smallest matrix is already too large for the L1 cache, we also measured the performance for much smaller matrices that have around 500 elements. We observed similar slowdowns.

```
; load vector elements
movsd (%rdi), %xmm0
mulsd 216(%rdi), %xmm1
movsd 248(%rdi), %xmm2
movsd 656(%rdi), %xmm3
movsd 664(%rdi), %xmm4
movsd 680(%rdi), %xmm5
movsd 1288(%rdi), %xmm6
; mult. with matrix values
mulsd (%rdx), %xmm0
mulsd 8(%rdx), %xmm1
mulsd 16(%rdx), %xmm2
mulsd 24(%rdx), %xmm3
mulsd 32(%rdx), %xmm4
mulsd 40(%rdx), %xmm5
mulsd 48(%rdx), %xmm6
; sum up the values
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm5, %xmm4
addsd %xmm2, %xmm0
addsd %xmm6, %xmm4
addsd %xmm4, %xmm0
addsd %xmm0, %xmm7
```

```
; load matrix values
movabsq $4599271452859079754, %r9   ; 3.10853173267728e-01
movd   %r9, %xmm0
movabsq $-4706094344638026474, %r10 ; -9.8909426205185e-07
movd   %r10, %xmm1
movabsq $-4692542897400292683, %r11 ; -7.9816150893424e-06
movd   %r11, %xmm2
movabsq $-4703078975711860500, %r12 ; -1.6276234691992e-06
movd   %r12, %xmm3
movabsq $-4701605526230882399, %r13 ; -1.9719284338375e-06
movd   %r13, %xmm4
movabsq $-4706094344638026474, %r14 ; -9.8909426205185e-07
movd   %r14, %xmm5
movabsq $-4706094344638026474, %r15 ; -9.8909426205185e-07
movd   %r15, %xmm6
; multiply with vector elements
mulsd (%rdi), %xmm0
mulsd 216(%rdi), %xmm1
mulsd 248(%rdi), %xmm2
mulsd 656(%rdi), %xmm3
mulsd 664(%rdi), %xmm4
mulsd 680(%rdi), %xmm5
mulsd 1288(%rdi), %xmm6
; sum up the values
addsd %xmm1, %xmm0
addsd %xmm3, %xmm2
addsd %xmm5, %xmm4
addsd %xmm2, %xmm0
addsd %xmm6, %xmm4
addsd %xmm4, %xmm0
addsd %xmm0, %xmm7
```

Figure 13: Left: code generated by `MLUnfolding` method. Right: code obtained when matrix values are set from immediate values instead of loading from the memory.

2.7 Combination of the Optimizations

We have presented five optimization ideas:

1. Reducing the memory offset values to have shorter instructions.
2. Restricting the `xmm` register set to `xmm0-xmm7`.
3. Creating a pool of distinct values.
4. Using vector instructions.
5. Embedding matrix values in the instructions.

Based on experimental results, we have seen that optimization idea (1) gives substantial speedup; (5) always gives significant slowdown; optimization (2) sometimes provides small amount of improvement; optimizations (3) and (4) give good speedup under certain conditions.

We have integrated optimizations (1) and (2) into our purpose-built compiler. Idea (5) is discarded. We have made optimizations (3) and (4) optional (recall that they do not go well together, though). Vectorization in the icc compiler can be turned off by passing the `-no-vec` flag; it is enabled by default in the `-O3` level of optimization. In Section 1.5, the time measurements were done using code compiled with the `-no-vec` flag. In this section we compare the performance obtained by icc, both with and without vectorization, to the performances that we are able to achieve. In this comparison, the following are the method names we use:

- `OurUnfoldingV1`: This is `MLUnfolding` together with optimizations (1) and (2).
- `OurUnfoldingV2`: This is `MLUnfolding` together with optimizations (1), (2), and (3).
- `OurUnfoldingV3`: This is the vectorized version of `OurUnfoldingV1`. That is, `MLUnfolding` together with optimizations (1), (2), and (4).

Tables 11 and 12 give the comparison of icc's output to ours when vectorization is *disabled*. We see that the quality of our output is on par with icc's. The last column in the tables show the ratio of icc-compiled code's time to our code's time; having a value greater than 1 means that our output is faster. On the average, our code is 1.07x the performance of icc's output. For 51 matrices out of 70, our code performs better than icc. Out of 19 matrices for which our code is worse, there are only 5 where we are not faster than `PlainSpMV`. So, in conclusion, we are able to generate code that can compete with icc's output, while avoiding the analyses and transformations that icc goes through.

Matrix	N	NZ	Dist. vals	Performances wrt `PlainSpMV` Unfolding	UnfoldingV2	OurUnfoldingV1	OurUnfoldingV2	Ratio of icc's best to our's best
dwt_419 (p)	419	1991	1991	1.72	1.73	2.61	1.89	**1.51**
str__600	363	3279	1972	1.29	1.15	1.44	1.26	**1.12**
minnesota (p)	2642	3303	3303	2.68	2.99	4.04	3.44	**1.35**
bcspwr06 (p)	1454	3377	3377	2.69	2.70	3.57	2.88	**1.32**
west0989	989	3518	1776	2.02	1.62	1.72	1.92	0.95
bfw398a	398	3678	92	1.23	0.89	1.08	1.37	**1.11**
bcsstk19	817	3835	1852	1.21	1.14	1.39	1.15	**1.15**
bcspwr08 (p)	1624	3837	3837	2.69	2.71	3.56	2.88	**1.31**
ck656	656	3884	3054	0.98	0.94	1.14	0.83	**1.17**
can__634 (p)	634	3931	3931	1.16	1.10	1.40	1.18	**1.21**
tub1000	1000	3996	1990	1.24	1.09	1.30	1.14	**1.05**
G33	2000	4000	2	2.37	1.25	1.80	2.56	**1.08**
bcsstk06	420	4140	1045	1.02	0.96	1.15	1.01	**1.13**
hor__131	434	4182	1553	1.06	0.88	1.11	1.06	**1.05**
gr_30_30	900	4322	2	4.04	1.01	1.32	1.82	0.45
pde900	900	4380	3248	2.74	0.94	1.31	1.08	0.48
cdde3	961	4681	5	3.50	0.84	1.33	1.28	0.38
bp__1600	822	4841	1803	1.86	1.46	1.93	2.01	**1.08**
email (p)	1133	5451	5451	1.55	1.66	2.01	1.62	**1.21**
steam2	600	5660	1071	0.93	0.88	1.10	1.00	**1.19**
gre_1107	1107	5664	11	1.82	1.21	1.57	1.84	**1.01**
fs_760_1	760	5739	4743	1.50	0.78	1.16	0.97	0.77
dwt_1242 (p)	1242	5834	5834	1.66	1.21	1.68	1.38	**1.01**
e05r0000	236	5846	1269	0.68	0.64	0.77	0.76	**1.14**
fpga_dcop_51	1220	5892	953	1.63	1.36	1.72	1.62	**1.06**
jpwh_991	991	6027	14	2.14	1.02	1.70	2.42	**1.13**
EVA (p)	8497	6726	6726	1.77	1.76	2.25	1.86	**1.27**
can_1072 (p)	1072	6758	6758	1.19	1.12	1.62	1.10	**1.36**
rdb1250	1250	7300	6	1.45	0.80	1.02	1.49	**1.03**
west2021	2021	7310	4235	1.61	1.22	1.67	1.54	**1.03**
mahindas	1258	7682	3291	1.28	0.85	1.01	1.10	0.86
GD06_Java (p)	1538	8032	8032	1.28	1.24	1.82	1.41	**1.42**
nos3	960	8402	149	1.04	0.71	0.82	0.90	0.87
blckhole (p)	2132	8502	8502	1.00	1.02	1.36	1.06	**1.32**
c-18	2169	8657	4861	1.42	1.06	1.29	1.16	0.91

Table 11: (Part 1 of 2) Comparison of the performance of the code we generate to the output of icc. Vectorization is disabled.

Matrix	N	NZ	Dist. vals	Performances wrt PlainSpMV: Unfolding	UnfoldingV2	OurUnfoldingV1	OurUnfoldingV2	Ratio of icc's best to our's best
tols4000	4000	8784	3188	2.61	0.96	1.27	1.19	0.49
pores_2	1224	9613	5407	0.80	0.64	0.87	0.85	**1.09**
spiral	1434	9831	3089	1.08	0.63	0.92	0.97	0.90
M80PI_n1	4028	9927	70	2.11	0.86	1.19	1.58	0.75
dw2048	2048	10114	693	2.02	0.85	1.23	1.25	0.62
watt__1	1856	11360	6524	0.88	0.84	1.14	1.04	**1.29**
watt__2	1856	11550	6589	0.83	0.84	1.12	1.00	**1.34**
bayer09	3083	11767	5003	1.09	0.86	1.22	1.13	**1.12**
Pd	8081	13036	432	2.80	1.78	2.27	3.04	**1.08**
add20	2395	13151	7390	1.04	0.89	1.25	1.12	**1.20**
lshp3466 (p)	3466	13681	13681	0.91	0.91	1.18	0.94	**1.30**
dwt_2680 (p)	2680	13853	13853	0.93	0.93	1.19	0.98	**1.27**
as-735 (p)	7716	13895	13895	1.83	1.84	2.40	1.98	**1.30**
orsreg_1	2205	14133	111	2.35	0.63	0.85	1.03	0.44
ca-GrQc (p)	5242	14496	14496	1.37	1.37	1.82	1.52	**1.33**
adder_trans_02	1814	14579	10327	0.85	0.78	1.00	0.92	**1.17**
bcsstk26	1922	16129	13480	0.84	0.79	0.98	0.86	**1.16**
plat1919	1919	17159	17120	1.06	0.61	0.80	0.66	0.76
wang2	2903	19093	1727	1.57	0.64	0.85	0.88	0.56
coater1	1348	19457	1380	0.69	0.52	0.70	0.83	**1.21**
add32	4960	19848	13883	1.17	1.07	1.38	1.20	**1.18**
olm5000	5000	19996	6	1.10	0.71	0.84	1.00	0.92
rw5151	5151	20199	150	2.25	0.63	0.89	1.09	0.48
sherman5	3312	20793	15096	0.70	0.69	0.81	0.74	**1.15**
saylr4	3564	22316	11	1.68	0.72	0.97	1.34	0.80
Oregon-1 (p)	11492	23409	23409	1.65	1.64	2.17	1.80	**1.32**
mcfe	765	24382	24381	0.41	0.40	0.55	0.48	**1.34**
lnsp3937	3937	25407	4176	0.69	0.63	0.78	0.74	**1.12**
fidap002	441	26831	11118	0.36	0.35	0.51	0.47	**1.40**
bcsstk14	1806	32630	14044	0.56	0.54	0.67	0.59	**1.19**
cavity05	1182	32632	3280	0.47	0.42	0.56	0.55	**1.19**
p2p-Gnutella04 (p)	10879	39994	39994	1.05	1.05	1.25	1.12	**1.19**
mbeause	496	41063	2100	0.44	0.34	0.51	0.71	**1.62**
cry10000	10000	49699	49599	1.96	0.62	0.78	0.65	0.40
mbeaflw	496	49920	19778	0.33	0.33	0.50	0.41	**1.49**

Table 12: (Part 2 of 2) Comparison of the performance of the code we generate to the output of icc. Vectorization is disabled.

Tables 13 and 14 give the comparison of icc's output to ours when vectorization is *enabled*. Here, we do not list `UnfoldingV2` because its vectorized version is never significantly better than vectorized `Unfolding`. When vectorized, we see that `PlainSpMV` has become noticeably fast. `Unfolding` is better than `PlainSpMV` for 39 matrices out of 70. There is substantial slowdown, in particular for the big matrices in our set. The code that we generate is still on par with icc-compiled code; on the average our performance is 1.01x the performance of icc's output. Our code is better than icc's for 45 of the matrices. A vectorization algorithm that is more sophisticated than the one we used in this work might bring better results. This is left as a future work.

Matrix	N	NZ	Dist. vals	Unfolding performance wrt PlainSpMV	OurUnfoldingV3 performance wrt PlainSpMV	Ratio of Unfolding to OurUnfoldingV3
dwt_419 (p)	419	1991	1991	1.05	1.59	**1.51**
str__600	363	3279	1972	0.87	0.97	**1.11**
minnesota (p)	2642	3303	3303	1.59	2.37	**1.49**
bcspwr06 (p)	1454	3377	3377	1.49	1.98	**1.33**
west0989	989	3518	1776	1.36	1.15	0.85
bfw398a	398	3678	92	1.00	0.88	0.87
bcsstk19	817	3835	1852	1.18	1.36	**1.15**
bcspwr08 (p)	1624	3837	3837	1.47	2.13	**1.45**
ck656	656	3884	3054	0.98	1.14	**1.16**
can__634 (p)	634	3931	3931	1.03	1.14	**1.10**
tub1000	1000	3996	1990	1.31	1.37	**1.04**
G33	2000	4000	2	2.09	1.60	0.77
bcsstk06	420	4140	1045	0.85	0.96	**1.12**
hor__131	434	4182	1553	0.92	0.96	**1.04**
gr_30_30	900	4322	2	4.55	1.34	0.29
pde900	900	4380	3248	3.34	1.34	0.40
cdde3	961	4681	5	3.93	1.34	0.34
bp__1600	822	4841	1803	1.05	1.09	**1.04**
email (p)	1133	5451	5451	0.84	1.08	**1.29**
steam2	600	5660	1071	0.82	0.95	**1.16**
gre_1107	1107	5664	11	1.37	1.18	0.86
fs_760_1	760	5739	4743	1.72	1.09	0.63
dwt_1242 (p)	1242	5834	5834	1.36	1.36	**1.00**
e05r0000	236	5846	1269	0.53	0.69	**1.29**
fpga_dcop_51	1220	5892	953	1.14	1.32	**1.16**
jpwh_991	991	6027	14	1.47	1.15	0.79
EVA (p)	8497	6726	6726	1.68	2.12	**1.26**
can_1072 (p)	1072	6758	6758	0.87	1.19	**1.38**
rdb1250	1250	7300	6	1.48	1.05	0.71
west2021	2021	7310	4235	1.11	1.13	**1.01**
mahindas	1258	7682	3291	1.11	0.81	0.73
GD06_Java (p)	1538	8032	8032	0.74	1.13	**1.52**
nos3	960	8402	149	0.98	0.78	0.79
blckhole (p)	2132	8502	8502	0.96	1.18	**1.23**
c-18	2169	8657	4861	1.13	1.01	0.90

Table 13: (Part 1 of 2) Comparison of the performance of the code we generate to the output of icc. Vectorization is enabled.

Matrix	N	NZ	Dist. vals	Unfolding performance wrt PlainSpMV	OurUnfoldingV3 performance wrt PlainSpMV	Ratio of Unfolding to OurUnfoldingV3
tols4000	4000	8784	3188	2.65	1.16	0.44
pores_2	1224	9613	5407	0.85	0.90	**1.07**
spiral	1434	9831	3089	1.04	0.78	0.75
M80PI_n1	4028	9927	70	1.72	1.14	0.66
dw2048	2048	10114	693	1.77	1.00	0.57
watt__1	1856	11360	6524	0.69	0.92	**1.34**
watt__2	1856	11550	6589	0.73	0.93	**1.27**
bayer09	3083	11767	5003	0.83	0.97	**1.17**
Pd	8081	13036	432	1.62	1.38	0.85
add20	2395	13151	7390	0.81	0.94	**1.15**
lshp3466 (p)	3466	13681	13681	0.80	0.96	**1.20**
dwt_2680 (p)	2680	13853	13853	0.79	0.94	**1.19**
as-735 (p)	7716	13895	13895	1.73	2.25	**1.30**
orsreg_1	2205	14133	111	2.27	0.87	0.38
ca-GrQc (p)	5242	14496	14496	1.36	1.79	**1.32**
adder_trans_02	1814	14579	10327	0.69	0.81	**1.17**
bcsstk26	1922	16129	13480	0.60	0.72	**1.20**
plat1919	1919	17159	17120	1.03	0.71	0.69
wang2	2903	19093	1727	1.59	0.84	0.53
coater1	1348	19457	1380	0.57	0.57	**1.00**
add32	4960	19848	13883	0.72	0.85	**1.18**
olm5000	5000	19996	6	1.14	0.87	0.77
rw5151	5151	20199	150	2.07	0.89	0.43
sherman5	3312	20793	15096	0.59	0.68	**1.16**
saylr4	3564	22316	11	1.78	0.92	0.51
Oregon-1 (p)	11492	23409	23409	1.44	1.90	**1.32**
mcfe	765	24382	24381	0.36	0.48	**1.34**
lnsp3937	3937	25407	4176	0.61	0.69	**1.12**
fidap002	441	26831	11118	0.33	0.46	**1.40**
bcsstk14	1806	32630	14044	0.46	0.55	**1.19**
cavity05	1182	32632	3280	0.42	0.50	**1.19**
p2p-Gnutella04 (p)	10879	39994	39994	0.89	1.06	**1.19**
mbeause	496	41063	2100	0.38	0.45	**1.17**
cry10000	10000	49699	49599	2.20	0.81	0.37
mbeaflw	496	49920	19778	0.30	0.44	**1.47**

Table 14: (Part 2 of 2) Comparison of the performance of the code we generate to the output of icc. Vectorization is enabled.

CHAPTER III

INTEGRATION OF THE OPTIMIZATIONS INTO THE COMPILER

We have presented various methods for generating unfolded code that impact the performance. Although these methods are designed with the unfolded spMV in mind, they are not dependent strictly on this context. The ideas we have presented may be applied in other contexts where long, straightline unfolded code is seen. It is, therefore, feasible to define the transformations in the form of compiler passes so that they can be reused. Currently, the transformations are implemented as part of our purpose-built compiler; hence they are not reusable in other contexts.

As a proof-of-concept that the transformations can be defined independent of the spMV context, we have defined the offset-reduction optimization (Section 2.2) as an LLVM [11, 12] pass. In this chapter we explain how this pass is implemented. We argue that the other optimizations can also be defined as compiler passes.

LLVM is a compiler infrastructure that features a three-phase design with (1) a frontend, (2) an optimizer, (3) a backend. There may be many different frontends for different programming languages. The responsibility of the frontend is to parse a program, written in some high-level programming language, to LLVM's intermediate representation (IR), which is much closer to the machine-level code and is independent of the source program's language. For a sample LLVM IR code, see Figure 14 where we give a snippet from the unfolded spMV code. The second phase of the compiler operates at the IR level. Here, many analyses and transformations optimize the IR-level code. This way, optimizations are reused for programs written in different programming languages. Once the IR-level optimizations are complete, the IR is given

```
define double @multByM(double* %v, double* %w, i64* %rows, i64* %cols, double* %vals) {
entry:
  store i64* %rows, i64** @rows1
  store i64* %cols, i64** @cols1
  store double* %vals, double** @vals1
  %0 = load double** @vals1
  %1 = getelementptr double* %0, i64 0
  %2 = load double* %1
  %3 = getelementptr double* %v, i64 0
  %4 = load double* %3
  %5 = fmul double %2, %4
  %6 = getelementptr double* %0, i64 1
  %7 = load double* %6
  %8 = getelementptr double* %v, i64 1
  %9 = load double* %8
  %10 = fmul double %7, %9
  %11 = getelementptr double* %0, i64 2
  %12 = load double* %11
  %13 = getelementptr double* %v, i64 30
  %14 = load double* %13
  %15 = fmul double %12, %14
  %16 = fadd double %5, %10
  %17 = fadd double %16, %15
  %18 = getelementptr double* %w, i64 0
  %19 = load double* %18
  %20 = fadd double %19, %17
  %21 = getelementptr double* %w, i64 0
  %store double %20, double* %21
  ...
```

Figure 14: The LLVM IR representation of naive unfolding of the spMV code.

to the backend of LLVM. The backend translates the IR to a target-specific format, such as ARM, X86, etc. Hence, for each target, there is a dedicated backend. Target-dependent analyses and optimizations are run in the backend. LLVM is specifically architected to have clear and well-defined application programming interfaces (API) between the three phases so that any of the phases can be used independently of the others in third-party projects.

LLVM provides compiler developers with a mechanism to write custom transformations. A transformation is called a *pass* in the LLVM terminology, because it makes a pass over the code to perform analyses/modifications. A custom pass needs to derive from the `Pass` class in the LLVM code base. There are several `Pass` classes defined for various needs. Each is defined as a Visitor [18]; they internally define

```
%RAX <def> = MOV64rm %RIP, 1, %noreg,  <ga:@vals1>, %noreg; mem:LD8[@vals1]
%XMM0 <def> = VMOVSDrm %RAX, 1, %noreg, 0, %noreg; mem:LD8[%1]
%XMM1 <def> = VMOVSDrm %RAX, 1, %noreg, 8, %noreg; mem:LD8[%6]
%XMM0 <def> = VMULSDrm %XMM0 <kill>, %RDI, 1, %noreg, 0, %noreg; mem:LD8[%3]
%XMM1 <def> = VMULSDrm %XMM1 <kill>, %RDI, 1, %noreg, 8, %noreg; mem:LD8[%8]
%XMM2 <def> = VMOVSDrm %RAX, 1, %noreg, 16, %noreg; mem:LD8[%11]
%XMM2 <def> = VMULSDrm %XMM2 <kill>, %RDI, 1, %noreg, 72, %noreg; mem:LD8[%13]
%XMM3 <def> = VMOVSDrm %RAX, 1, %noreg, 24, %noreg; mem:LD8[%16]
%XMM3 <def> = VMULSDrm %XMM3 <kill>, %RDI, 1, %noreg, 80, %noreg; mem:LD8[%18]
%XMM4 <def> = VMOVSDrm %RAX, 1, %noreg, 32, %noreg; mem:LD8[%21]
%XMM4 <def> = VMULSDrm %XMM4 <kill>, %RDI, 1, %noreg, 144, %noreg; mem:LD8[%23]
%XMM5 <def> = VMOVSDrm %RAX, 1, %noreg, 40, %noreg; mem:LD8[%26]
%XMM5 <def> = VMULSDrm %XMM5 <kill>, %RDI, 1, %noreg, 152, %noreg; mem:LD8[%28]
%XMM0 <def> = VADDSDrr %XMM0 <kill>, %XMM1 <kill>
%XMM0 <def> = VADDSDrr %XMM0 <kill>, %XMM2 <kill>
%XMM0 <def> = VADDSDrr %XMM0 <kill>, %XMM3 <kill>
%XMM0 <def> = VADDSDrr %XMM0 <kill>, %XMM4 <kill>
%XMM0 <def> = VADDSDrr %XMM0 <kill>, %XMM5 <kill>
%XMM0 <def> = VADDSDrm %XMM0 <kill>, %RSI, 1, %noreg, 0, %noreg; mem:LD8[%36]
VMOVSDmr %RSI, 1, %noreg, 0, %noreg, %XMM0 <kill>; mem:ST8[%39]
```

Figure 15: A snippet from LLVM's machine-dependent representation for the naive unfolding of the spMV code, where the target machine is X86_64.

the mechanism to traverse the code components. For instance, there is a `Pass` class that traverses the functions in a module, there is one that traverses basic blocks in a function, and yet another that goes over the instructions in a basic block. To write a custom pass, one needs to subclass a `Pass` class chosen according to the purposes of the custom pass. Then, the visit method[1] should be overridden to define the specific behavior of the pass.

In LLVM, custom passes can be written and plugged into the compiler both in the second phase (i.e. the IR optimizer) and the third phase (i.e. the backend). In our case, the offset-reduction optimization is dependent on the assembly-level code. Hence, we wrote a pass as part of the third phase[2]. This way, we were able to operate on the machine-dependent representation of the code. Figure 15 shows a snippet of LLVM's machine-dependent representation of the unfolded spMV code.

Our offset-reduction optimization pass operates in three phases. In the first phase,

[1]For a FunctionPass, this method is called `runOnFunction`.
[2]In LLVM terminology, we wrote a `MachineFunctionPass`.

we do the following:

- All the basic blocks in the machine-level representation of functions of the source code are traversed.

- The traversal happens in post-order according to the Strongly-Connected Component (SCC) ordering of the basic blocks. In the case of unfolding, because there are no loops, there is one big basic block. But our pass would still work if there were cycles in the control flow graph of the code.

- Our pass goes over the instructions. When we come across a memory access instruction whose memory address operand is a register with an immediate constant, we record the instruction in a hashtable where the key is the register.

After all the instructions are traversed, the second phase of our pass takes place. At this phase we analyze the hashtable that comes from the first phase as follows:

- For each register in the hashtable, the recorded instructions are analyzed.

- We look at instruction intervals where the register is alive. Within these intervals, we look for patterns where the memory offsets monotonically increase with 8-byte increments.

Finally, in the third phase, we operate on each pattern detected in the second phase. We insert LEAQ instruction in appropriate places to increment the value of the base register, and we adjust the memory offsets accordingly.

In Table 15 we show the performance and code size with respect to naive unfolding after applying the offset-reduction pass.

Matrix	Performance	Code size	Matrix	Performance	Code size
dwt_419 (p)	1.18	0.87	tols4000	1.08	0.91
str__600	1.11	0.89	pores_2	1.18	0.87
minnesota (p)	1.08	0.91	spiral	1.08	0.91
bcspwr06 (p)	1.09	0.90	M80PI_n1	1.11	0.89
west0989	1.10	0.88	dw2048	1.12	0.88
bfw398a	1.14	0.88	watt__1	1.11	0.88
bcsstk19	1.12	0.89	watt__2	1.13	0.88
bcspwr08 (p)	1.09	0.89	bayer09	1.11	0.89
ck656	1.14	0.87	Pd	1.07	0.91
can__634 (p)	1.15	0.88	add20	1.14	0.89
tub1000	1.12	0.88	lshp3466 (p)	1.13	0.88
G33	1.09	0.90	dwt_2680 (p)	1.12	0.88
bcsstk06	1.13	0.88	as-735 (p)	1.08	0.90
hor__131	1.12	0.88	orsreg_1	1.12	0.87
gr_30_30	1.13	0.88	ca-GrQc (p)	1.08	0.89
pde900	1.11	0.88	adder_trans_02	1.10	0.89
cdde3	1.13	0.88	bcsstk26	1.10	0.88
bp__1600	1.11	0.89	plat1919	1.09	0.87
email (p)	1.13	0.88	wang2	1.10	0.88
steam2	1.14	0.87	coater1	1.08	0.90
gre_1107	1.13	0.88	add32	1.17	0.88
fs_760_1	1.14	0.88	olm5000	1.10	0.88
dwt_1242 (p)	1.14	0.88	rw5151	1.08	0.88
e05r0000	1.12	0.90	sherman5	1.10	0.88
fpga_dcop_51	1.13	0.89	saylr4	1.10	0.88
jpwh_991	1.20	0.88	Oregon-1 (p)	1.07	0.90
EVA (p)	1.08	0.91	mcfe	1.07	0.91
can_1072 (p)	1.16	0.87	lnsp3937	1.12	0.88
rdb1250	1.15	0.88	fidap002	1.08	0.92
west2021	1.15	0.89	bcsstk14	1.10	0.89
mahindas	1.15	0.90	cavity05	1.07	0.90
GD06_Java (p)	1.20	0.89	p2p-Gnutella04 (p)	1.08	0.88
nos3	1.19	0.87	mbeause	1.05	0.93
blckhole (p)	1.14	0.89	cry10000	1.10	0.88
c-18	1.12	0.89	mbeaflw	1.06	0.94

Table 15: Performance and code size with respect to naive unfolding after applying our offset-reduction pass.

CHAPTER IV

CONCLUSION

Specialization of sparse matrix-vector multiplication code according to the matrix may bring significant performance improvements. A method of specialization is to fully unfold the code. In this work, we have experimentally investigated the performance of unfolded spMV code using real-world matrices. We have shown that

- substantial speedup can be obtained by unfolding;
- the quality of an industry-strength compiler can be achieved by manual generation of assembly-level code together with low-level optimizations. This way, code generation can take place much more rapidly as compared to using a general-purpose compiler.

We have discussed five possible low-level optimizations; four of these speed up the code significantly under certain conditions. Finally, we have defined one of the optimizations as a code-transforming pass. This is a proof-of-concept that the optimizations can be defined modularly. to allow applying them in contexts other than fully unfolding the spMV code.

Bibliography

[1] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Towards realistic performance bounds for implicit CFD codes," in *Proceedings of Parallel CFD'99*, pp. 241–248, 1999.

[2] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, vol. 50, no. 1, pp. 36–77, 2009.

[3] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2011.

[4] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, pp. 247–256, Aug 2009.

[5] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 18:1–18:11, ACM, 2009.

[6] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: An extended compression format for spmv on shared memory systems," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 247–256, ACM, 2011.

[7] D. Guo and W. Gropp, "Optimizing sparse data structures for matrix-vector multiply," *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 115–131, Feb. 2011.

[8] S. Kamin, M. J. Garzarán, B. Aktemur, D. Xu, B. Yılmaz, and Z. Chen, "Optimization by runtime specialization for sparse matrix-vector multiplication," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, (New York, NY, USA), pp. 93–102, ACM, 2014.

[9] A. H. Sameh and V. Sarin, "Hybrid parallel linear system solvers," *International Journal of Computational Fluid Dynamics*, vol. 12, no. 3-4, pp. 213–223, 1999.

[10] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan, "Shonan challenge for generative programming: short position paper," in *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, PEPM '13, (New York, NY, USA), pp. 147–154, ACM, 2013.

[11] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*, (Washington, DC, USA), IEEE Computer Society, 2004.

[12] "The llvm compiler infrastructure." http://llvm.org.

[13] N. Johnson, "Code size optimization for embedded processors," Tech. Rep. UCAM-CL-TR-607, University of Cambridge, 2004.

[14] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua, "In search of a program generator to implement generic transformations for high-performance computing," *Science of Computer Programming*, vol. 62, no. 1, pp. 25 – 46, 2006. Special Issue on the First MetaOCaml Workshop 2004.

[15] R. Davies and F. Pfenning, "A modal analysis of staged computation," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 258–270, ACM, 1996.

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

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

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing, 1995.

VITA

I obtained my BSc degree from Ege University, Department of Computer Engineering. I'm currently working as a software engineer at ING Bank.