OPTIMIZATION BY RUNTIME SPECIALIZATION FOR SPARSE MATRIX-VECTOR MULTIPLICATION

BY

DANQING XU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisor:

Professor María Jesús Garzarán

Abstract

Runtime specialization optimizes programs based on partial information available only at run time. It is applicable when some input data is used repeatedly while other input data varies. This technique has the potential of generating highly efficient codes.

In this thesis we explore the potential for obtaining speed-ups for sparse matrix-dense vector multiplication using runtime specialization, in the case where a single matrix is to be multiplied by many vectors. We experiment with five methods involving run-time specialization with parallelization, comparing them to methods that do not (including Intel's MKL library). For this work, our focus is the evaluation of the parallel speed-ups that can be obtained with runtime specialization without considering the overheads of the code generation.

Our experiments run on four different machines with 88 matrices from the Matrix Market and Florida collections, among others. In 348 of those 352 cases, the specialized code runs faster than any version without specialization. In the worst case, the specialized code is 7 percent slower than the Intel's MKL library. If we only use specialization, the average speedup with respect to Intel's MKL library ranges from 1.416x to 1.470x, depending on the machine. We have also found that the best method depends on the matrix and machine; no method is best for all matrices and machines [1].

.

[1] This material is based upon work supported by the National Science Foundation under Award CCF 1017077

To Father, Mother and Barbara.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The technique of `program specialization` begins with the observation that many computations get their inputs in two parts: an early, stable part, and a late, dynamic part. One then asks the question: Given the early data, can we fashion a new, specialized, program that will process the dynamic data very efficiently? For example, in some numerical applications, a single matrix M is multiplied by many vectors v; M is early and stable, the vectors late and dynamic. Can we create a very efficient function $\texttt{multBy}_M(v, w)$ to multiply M by an input vector v and place the result in w?

Program specialization is a well-studied area [1, 2, 3]. Research has produced many examples of programs, in many problem domains, that have been optimized by specialization. However, most of the work has focused on languages and infrastructure, rather than realistic applications. Take the matrix multiplication example again. The "optimal" approach is simply to unfold the calculation. Instead of a loop iterating over M and v, $\texttt{multBy}_M$ consists of a long sequence of assignment statements of the form

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

where the italicized parts — i, M_{i,j_0}, j_0, etc. — are *fixed* values, not variables or subscripted arrays. (The simpler case of vector-vector dot product is a standard "toy" example in this field [4]; a variation of sparse matrix-vector multiplication was recently posed as a Shonan Challenge [5]). This code is "optimal" in the sense of producing the minimum instruction count; a standard Compressed-Sparse-Row (CSR) loop (see Section 2.2) will execute perhaps five times as many instructions as this unfolded code. They will, of course, execute the same number of floating-point operations; the additional instructions are all integer, control, or load operations.

However, it will come as no surprise to those who work in the area of high-performance computing that instruction count tells only a part of the story. Execution speed is affected by such factors as the quality of the code (e.g. register usage), and memory system performance. Traditionally, the latter is concerned primarily with avoiding cache misses when accessing `v` and `w` (with accesses to M being purely sequential and therefore not subject to optimization); a new concern that arises here is access to the code itself.

This work addresses the potential for optimizing parallel sparse matrix–dense vector multiplication by specialization relative to the matrix M, using matrices of realistic size and structure. To that end, we explore a variety of methods and report on their efficiency. The methods (described in detail in Section 2) are these:

Compressed sparse row (`CSR`). This is the straightforward implementation using the most traditional representation for sparse matrices. Some efficiency is gained by unrolling the inner loop; we refer to `CSR` with the inner loop unrolled u times as $\texttt{CSR}_u$.

Unfolding. This is the simple unfolded code described above.

CSRbyNZ. This method generates a loop for each group of rows that contain a given number of non-zeros [6]. In effect, this provides a perfect unrolling of the inner loop of `CSR`.

Stencil. This method analyzes the matrix to find the patterns of non-zero entries in each row of M, and generates, for each pattern, a loop that handles all the rows that have that pattern.

GenOSKI. This method analyzes the matrix to find the patterns of non-zero entries in each block of size $r \times c$, and for each pattern generates straight-line code [7]. A motivation of this method is to avoid the zero-fill problem of OSKI [8], that generates efficient per-block code by inserting some zeros into the matrix data.

We tested all the methods on 88 matrices and 4 different machines. Most of the matrices are from the Matrix Market [9] or Florida collections [10, 11]. A few are matrices obtained from the discretization of a Poisson problem and used as GPU SpMV data sets [12, 13]. Our experimental results show the two main points of this work:

1. Speed-ups can be obtained by runtime specialization. In most cases, one of the methods involving runtime code generation is the fastest.

2. There is no one best method: it varies both across machines and across matrices.

Specifically, out of our 352 (88×4) trials, the best specializers were: `Stencil` (50), `GenOSKI` (38), `Unfolding` (98), `CSRbyNZ` (61), `CSR` (101), and baseline `MKL` (4).

We compare our results with three state of the art libraries: the Intel `MKL` library, `BiCSB` [14], and `CSX` [15]. `BiCSB` [16] is implemented on top of CSB [17], a new parallel sparse matrix data structure that allows efficient SpMV on multicores. `BiCSB` requires some restructuring of the data, but no runtime generation of the code. `CSX` [18] is based on the Compressed Sparse eXtended (CSX) format that allows for a flexible storage format to support a variety of patterns within the sparse matrix, such as horizontal, vertical, diagonal, anti-diagonal, or blocks [1].

We can classify all libraries in three groups: those that are completely generic and operate on the standard CSR representation (`CSR`); those that require some restructuring of the data but no runtime generation of code (`CSB`, `BiCSB`, and OSKI); and those that require runtime code generation (`Unfolding`, `CSRbyNZ`, `Stencil`, `GenOSKI`[2], and `CSX`). The distinction matters because it refers to the latency of each method — the preparation time needed before a method can report its first result. $\texttt{CSR}_u$ has zero latency, and methods that only restructure the data have lower latency than methods that generate code. Of course, latency varies widely *within* the latter two categories as well.

The main contribution of this thesis is a systematic comparison of a number of methods for performing sparse matrix–dense vector multiplication, including methods that are specialized to a particular matrix. The methods evaluated are "generic" in the sense that they are not designed for matrices of any very particular form, but would apply in general to sparse matrices of the kind found in the Matrix Market [9] and Florida Sparse Matrix Collection [10, 11].

We discuss some of the reasons for the timings we are seeing, including matrix characteristics, the effect of code and data size and cache size and the machines configuration. In addition, we explain how this work fits into the overall goal of creating a matrix-vector multiplication library.

The structure of the thesis is this: Chapter 2 describes in detail the methods we are studying for performing matrix-vector multiplication; most involve code generation. Chapter 3 discusses some aspects of the methods that affect performance. Chapter 4 describes our experimental setup, including the machines on which we have run our tests and the matrices we used; Chapter 5 shows our performance numbers. In Chapter 6, we discuss how this work might find applications in practice, the central issue being how to deal with latency. Chapter 7 discusses related work; conclusions are presented in Chapter 8.

[1] These methods are only ran for 23 of the 88 matrices. Due to some library conflicts, CSX only runs on two of the four machines.

[2] Potentially, the code for any possible pattern of `GenOSKI` can be generated off-line; however, because there are too many possibilities (e.g. 2^{16} when using 4×4 blocks), opting for runtime generation is likely to be more feasible for this method.

Chapter 2

Methods

In this section, we describe the methods we use. In this discussion, we assume M is an $n \times n$ matrix, with nz non-zeros. We use zero-based indexing for all arrays. The code shown in this section is drawn from the actual generated code. After discussing the methods, in Sections 3.1 through 3.4, we discuss some aspects of the methods that seem likely to affect performance; we will return to these in Section 5, after seeing the actual timings.

2.1 Compressed Sparse Rows (CSR)

The most common representation for sparse matrices is *Compressed Sparse Rows* (`CSR`). It consists of three arrays:

- `mvalues` is an array of floating-point numbers of length nz containing the non-zero values of M in row-major order.
- `cols` is an integer array of length nz. Element i of this array contains the column number of the i^{th} element in the `mvalues` array.
- `rows` is an integer array of length $n+1$. Element j of this array gives the `mvalues`-index of the first non-zero element of row j.

With this representation, a standard `CSR` loop looks as follows (recall that `v` is the input vector, `w` is the output vector):

```
for (i = 0; i < n; i++){
  ww = 0.0;
  k = rows[i];  // mvalues[k] = M[i,cols[k]],
                // the first non-zero in row i
  for (; k < rows[i+1]; k++)
    ww += mvalues[k] * v[cols[k]];
  w[i] += ww;
}
```

2.2 CSR Unrolling

$\texttt{CSR}_u$ partially unrolls the inner loop of the standard `CSR` method u times. This method requires the addition of a "clean-up" loop handling the leftover elements. The data layout is identical to `CSR`. Unrolling can produce more efficient code than `CSR` due to additional instruction level parallelism and reduced loop overhead. However, the difference in performance between `CSR` and $\texttt{CSR}_u$ is expected to be small. In reality, our experiment shows that $\texttt{CSR}_1$ (without unrolling) generally performs better than any higher level of unrolling, because the compiler can do very well in loop unrolling nowadays.

2.3 CSRbyNZ

This method groups the rows of M according to the number of non-zeros they contain, and generates one loop for each group. The array `rows` contains a permutation of the row numbers, in which all the rows with a particular non-zero count are grouped together; `cols` and `mvalues` serve the same purpose as with `CSR`. So, for example, if there are exactly six rows of M that have three non-zeros, the loop for those rows would be:

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

Here, `a` indexes over `rows` and `b` indexes over `mvalues`. `mvalues` contains the non-zeros of M in the order in which they are consumed by these loops.

This method gains its efficiency from long basic blocks in each loop, which can be compiled efficiently. It provides, in effect, a perfect unrolling of the inner loop of `CSR`. (`CSRbyNZ` is similar to the method described by Mellor-Crummey and Garvin [6].)

2.4 Unfolding

`Unfolding` completely unfolds the `CSR` loop and produces a straight-line program, Despite its simplicity, it needs a detailed explanation as the code it generates has interesting and important implications on the binary code produced by the compiler.

First, recall that this method generates a statement per each matrix row i in the following way:

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

In principle as well as in practice, this method produces the lowest number of dynamic instructions. However, it also produces, by far, the longest code. Indeed, from a memory point of view, it provides an extremely wasteful encoding of the basic data needed for this calculation. Yet, surprisingly, in our tests, we have seen that `Unfolding` occasionally beats the other methods substantially, even for very large matrices. The reason for this is that many matrices have *repeated values*; indeed, the number of distinct values in our sample matrices is usually much less than nz (see Table A.2).

This produces speed-ups for two reasons: reduced memory load, and reduced instructions because of common subexpressions. To see this, suppose there are only three distinct values in the matrix (say, 3, 5, and 9) and let the first two lines of the generated code be

```
w[0] += 9*v[2] + 9*v[3] + 5*v[8] + 3*v[9];
w[1] += 5*v[8] + 3*v[9] + 9*v[11];
```

Having a nonzero value repeated on the *same row* of the matrix allows applying anti-distribution of multiplication over addition (i.e. $c \times v_i + c \times v_j = c \times (v_i + v_j)$). Having the same value repeated on the *same column* of the matrix enables common subexpression elimination (CSE). After applying both optimizations, the above code would look like this:

```
double temp = 5*v[8] + 3*v[9];
w[0] += 9*(v[2] + v[3]) + temp;
w[1] += temp + 9*v[11];
```

The floating point constants are emitted by the compiler — we examined `icc`, `gcc`, and `clang` — into the data section of the object code, and loaded into registers. When the distinct values are very few, registers can be reused to reduce memory loads. In effect, the code above can be compiled as if it were:

```
double M[3] = {9, 5, 3};
double temp = M[1]*v[8] + M[2]*v[9];
double m9 = M[0];
w[0] += m9*(v[2] + v[3]) + temp;
w[1] += temp + m9*v[11];        // m9 reused
```

Unlike all our other methods, and contrary to what we said in the introduction, specialization by this method actually allows a *reduction in the number of floating point operations.*

It is worth mentioning that, although the number of distinct values is usually much less than nz, this fact alone is not that helpful; the number has to be small enough that we are likely to see many repeated values in each row and column, thus allowing the optimizations described. Furthermore, by causing references to matrix values to be accessed out of order — in all other methods, these values are stored in an array that is accessed in strictly sequential order — these optimizations can have a negative effect on locality.

2.5 Stencil

Where `CSRbyNZ` divides up the rows of M according to the number of non-zeros, `Stencil` divides them up according to the exact pattern of non-zeros in a row. Specifically, the "stencil" of each row is defined as the location of non-zeros relative to the main diagonal. So, if row r has non-zeros in columns $r-1$, r, $r+1$, and $r+3$, its stencil would be $\{-1, 0, 1, 3\}$. All the rows that have the same stencil can be handled in a single loop. For example, if rows 2, 4, and 6 are the only ones with stencil $\{-1, 0, 1, 3\}$, then the loop for this stencil is shown below, where the values of M are laid out in the order in which they are consumed by these loops:

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

Notice that if a stencil pattern has only one row, the loop can be eliminated by using the inner block of the loop:

```
w[8385] += mvalues[0] * vv[-1] + mvalues[1] * vv[0] + mvalues[2] * vv[1]  + mvalues[3] * vv[3];
```

Like `CSRbyNZ`, `Stencil` gets its efficiency from the long basic blocks inside each loop. But `Stencil` also gains an advantage in memory accesses, because it entirely eliminates the `cols` array and the indirect access to `v`. Thus, for matrices with a modest number of stencils, this method can be the most efficient. However, when the matrix has many stencils, the code size can get quite large, reducing its efficiency.

2.6 GenOSKI

This method is based on OSKI [8, 19, 20] and is similar to PBR [7]. The idea of OSKI is to divide the matrix into dense blocks (of size, say, $b \times b$) and perform the multiplication on a block basis. By having a loop whose body handles blocks of size $b \times b$, the goal of this optimization is to increase register reuse. It may also reduce the amount of memory required to store indices for the matrix M, since a single pair of indices is stored per block. (For example, if all blocks were perfectly dense, arrays `rows` and `cols` would each be of length nz/b^2, for a total size of $2nz/b^2$, as compared to the total size of $nz + n$ for these arrays in `CSR`.)

The drawback of OSKI is that non-empty blocks may still contain zeros, and those have to be added to M explicitly. This increases both the number of floating-point operations and memory communication. This zero fill substantially determines whether this method will be efficient. Our experience shows that 1×2, 2×1, and 2×2 blocks are occasionally efficient, but larger blocks almost never are. `GenOSKI` is our attempt to overcome the zero fill problem by generating code.

`GenOSKI` has one loop for each *block pattern* of non-zeros in this matrix. For each pattern, two arrays hold the list of "block locations," the indices of the northwest corner of the blocks that have that pattern. For example, consider a matrix divided into 3×3 blocks and having 18 blocks conforming to the pattern of non-zeros 1,1,0; 1,1,1; 0,1,1: the first two columns on row 0; all three columns on row 2; the second and third columns on row 3. The loop to handle these 18 blocks is shown below and the pattern is shown in Figure 2.1. Here `a` and `b` are global variables indexing over blocks and over values, respectively.

```
for (i = 0; i < 18; i++, a++) {
  ww = w + rows[a];
  vv = v + cols[a];
  ww[0] += vv[0]*mvalues[b] + vv[1]*mvalues[b+1];
  b += 2;
  ww[1] += vv[0]*mvalues[b] + vv[1]*mvalues[b+1]
        +  vv[2]*mvalues[b+2];
  b += 3;
  ww[2] += vv[1]*mvalues[b] + vv[2]*mvalues[b+1];
  b += 2 ;
}
```

`GenOSKI` has low overhead, and indeed often performs well, especially when most blocks are fairly dense. This is a bit surprising, because there are many reasons it should not do so. Zero fill is not a problem *per se*, but it does have an impact: we need to maintain two indexes per block (stored in the arrays `rows`

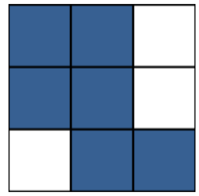

Figure 2.1: Example of genOSKI 3 pattern with six non-zero entries in a block.

and `cols`), so if there are many sparse blocks, this entails more data than `CSR`. Furthermore, `GenOSKI` can potentially generate a lot of code: for 4×4 blocks, there are 65,535 distinct patterns, which means every 4×4 blocks could have a different pattern. In practice, the number of patterns in a matrix is much smaller than the maximum (Table A.2). Lastly, unlike all the other methods, `GenOSKI` does not calculate entire rows at a time, which means that, where the other methods do a single write to each element of `w` — so exactly n writes — `GenOSKI` may do as many as n/b reads *and* writes for each row, or a total of nz/b memory operations on `w`. Nonetheless, as we have noted and will see in Section 5, it often does quite well.

Chapter 3

Performance Issues

In this section we discuss some aspects of the methods that are likely to affect performance.

3.1 Memory Requirements

A significant difference between specialized methods and "generic" methods is that specialization can produce large codes, which can in turn have a major impact on performance. On the other hand, by folding data into the code, the non-code data storage requirements can be reduced. Table 3.1 contains the expressions to compute code and data size for the various methods. Here we provide some explanation of that table.

$\texttt{CSR}_u$**:** Code size of $\texttt{CSR}_u$ is constant, and, for the values of u we consider, small. Data consists of array `mvalues` containing the non-zeros of M (nz doubles); array `rows` containing indices into the `cols` array (n integers); and the `cols` array giving the column of each non-zero (nz integers). (Due to a technicality of the representation, `rows` is actually of length $n + 1$.)

`CSRbyNZ`**:** Since a different loop is generated for each group of rows with the same count of non-zeros, the code size for `CSRbyNZ` is a function of the number of distinct non-zero counts (Row_nz), as well as the number of non-zeros in each group (nz_row_i). In practice, Row_nz is usually small (Table A.2), so code size is modest. Data size is similar to`CSR` except that `CSRbyNZ` doesn't take care of the empty rows that reduces the rows array in data.

`Unfolding`**:** For most matrices, `Unfolding` produces very much the longest code of any of our methods. (In rare cases, `Stencil` can produce code as long; no other method comes close.) As discussed above, repeated values can allow for optimizations that, in some cases, can significantly reduce code size, but this is rare, and in any case still leaves the code very long. (At the very least, the size of the code is $O(n)$, since there is one assignment for each row.) As far as data size, repeated elements reduce this significantly in many cases.

	CSR	CSRbyNZ	Unfolding	Stencil	GenOSKI
Code Size	c	$\sum_{i=1}^{Row\_nz} nz\_row_i * c_1$ $Row\_nz * c_2$	(possibly) $nz * c$	$\sum_{i=1}^{stencils} nz\_stencil_i * c_1$ $stencils * c_2$	$\sum_{i=1}^{patterns} nz\_pattern_i * c_1$ $patterns * c_2$
Data Size	$nz * 8+$ $nz * 4+$ $(n + 1) * 4$	$nz * 8+$ $nz * 4+$ $ner * 4$	$distinct\_nz * 8$	$nz * 8+$ $ner * 4$	$nz * 8+$ $nblocks * (4 + 4)$

Table 3.1: Expressions to compute Code and Data size for the different methods. `‘‘ner’’` is non-empty rows.

`Stencil`: The code size of this method depends on the number of stencils and the size of each stencil. As shown in Table A.2, the number of stencils varies widely from matrix to matrix.

`GenOSKI`: The code size for `GenOSKI` is primarily a function of the number of distinct patterns that appear in the matrix. As with stencils, this number varies widely from matrix to matrix (Table A.2). In practice, it is always smaller, and usually *much* smaller, than the number of stencils.

3.2 Memory Reference Locality

Another issue affecting performance that will vary by method is locality of memory references. All of our methods except `Unfolding` maintain the values of M in an array of length nz and access it sequentially; there is nothing to be done here about locality. Similarly, the location data in `rows` and `cols` are accessed sequentially. The issue of locality shows up in how the methods reference the input and output vectors `v` and `w`.

`CSR`: `CSR` maintains perfect locality relative to `w`, as it assigns to its elements sequentially. If M is strongly banded — meaning the non-zeros are clustered around the main diagonal — then it will have good locality in `v` as well. In most cases, there is a dense cluster of non-zeros around the main diagonal, but also a good number of non-zeros elsewhere; in this case, access to `v` will begin to look random, and locality will be poor.

`CSRbyNZ`: Here, because of the reordering of rows, access to `w` is no longer sequential. Furthermore, any "natural" locality in `v` — as when a matrix is strongly banded — may be lost. As a consequence, this method does not have particularly good memory behavior relative to either `v` or `w`.

`Stencil`: Memory access behavior of `Stencil` is similar to `CSRbyNZ`. Because each stencil loop may cover rows that are randomly distributed throughout M, and also each stencil contains elements of M potentially randomly distributed throughout a single row, accesses to `v` and `w` are arbitrary.

`GenOSKI`: As with all other methods, although `GenOSKI` appears to access data "out of order," the access to the values and to `rows` and `cols` are again perfectly sequential. However, as with `CSRbyNZ` and `Stencil`, accesses to `v` and `w` bear no obvious relation to the natural order, and are likely to be highly non-localized. (Aside from locality issues, we noted earlier that `GenOSKI` performs many more memory operations relative to `w` than the other methods.)

3.3 Parallelization

In this work, we run all of our codes in parallel. It is also interesting to see how these methods perform sequentially, but most researchers are using parallel codes, so that parallel times are easier to compare to other methods. For example, we have found that MKL does not perform very well in sequential mode, so that without running it in parallel, comparisons are fundamentally unfair. As another example, `CSX` does not claim to have good performance in the sequential case, but only when parallel execution creates memory contention.

Parallelization of these codes is generally quite straightforward. It is just a matter of splitting M into four horizontal tranches, with approximately equal numbers of elements, applying our methods to each, producing four functions to be run on the four cores. For `CSR` and `Unfolding`, there is really nothing more to it. We split M into four tranches in the obvious way (what we call "split-by-count") for these two methods.

For `CSRByNZ`, `Stencil`, or `GenOSKI`, there is one choice to be made before doing the split, and that is whether to group the rows before splitting. Consider `Stencil`: Suppose M has s stencils, and they are spread throughout the matrix. If we split M by "split-by-count", we are likely to have all s stencils, more or less, show up in each tranche; if there are a lot of stencils, the code running on each processor will be large. If instead we first group the rows of M by stencil and *then* do the split into four pieces (we call this "split-by-pattern"), each piece will have only a portion of the stencils and will therefore have less code, which is generally better for performance. Note that, for `stencil` and `CSRbyNZ`, we already have to sort the rows into groups, so split-by-pattern is no extra work.

`GenOSKI` presents a somewhat different problem. The method divides the matrix up by patterns, and handles every occurrence of a given pattern in a single loop. If we generate this code first, then assign a subset of the loops to each core, it gives us an even split *and* minimizes code size. However, there is a problem alluded to earlier: any of the patterns can contribute values to any of the rows causing the race condition problem. If we had code running on separate cores reading and writing to the same location in `w`, we would have to put locks on each one. On the other hand, if we split M into tranches (split-by-count),

		loome2	loome3	i2pc3	i2pc5
genOSKI4	avg. speedup	1.28	1.09	1.80	2.16
	max speedup	1.98	2.14	3.20	4.44
genOSKI5	avg. speedup	1.28	1.06	2.00	2.39
	max speedup	2.08	2.03	3.60	5.01

Table 3.2: Speedup of using split-by-count vs split-by-pattern.

and generate (sequential) `GenOSKI` code separately in each tranche, there is no need for locks. Although split-by-count results in larger code, the effect is more than offset by avoiding the need for locking [1].

Table 3.2 shows the speedup of split-by-count over split-by-pattern for genOSKI4 (4×4 genOSKI block) and genOSKI5 (5×5 genOSKI block) methods and four machines. The table shows the average and maximum speedup for each platform and method. The average speedup ranges from 1.058 to 2.388. Accordingly, we parallelize `GenOSKI` using split-by-count.

3.4 Load Balancing

We have observed a problem of load imbalance for large matrices. To address that issue, we have followed the following strategy. Our code generator estimates the cost of each piece of code by counting the dynamic number of flops the code executes. It then produces a list of functions with similar amount of work (dynamic flops). We produce more functions that number of threads, so that we can evenly partition these small functions among the threads to obtain load balance. However, for matrices with large loops, this strategy does not work well. Thus, when the cost of a loop is greater than a certain threshold (that the programmer specifies), we split the loop into n equal loops, where n is the number of threads running the computation. Each loop is placed in a different function, and each function will then be assigned to a different thread.

This loop splitting approach alleviates the imbalance problem, but does not completely fix it for all the matrices. Thus, in addition, to loop distribution, we also use "randomization". The idea is that after splitting the large loops among the threads, the rest of the loops (or statements) are placed in functions, and then the functions are randomly assigned to the threads. To avoid destroying locality, rather than randomizing individual functions, we randomize blocks of consecutive functions. Our experimental results show that blocks of 32 consecutive functions, where each function contains approximately about 500 flops, produce relatively better results.

We have assessed the impact of this strategy by running all the matrices and three methods (`Stencil`, `CSRbyNZ` and `Unfolding`) with and without it. `genOSKI` cannot use this strategy, as this could cause a data

[1] Notice that it is possible to parallelize GenOSKI using split-by-pattern without locks by locally accumulating the partial results of the output vector and later performing a global reduction, but we did not implement this code version.

		loome2	loome3	i2pc3	i2pc5
`Stencil`	# matrices is better	7	4	17	11
	# matrices is worse	0	0	0	0
	Avg. Speedup	1.03	1.02	1.08	1.05
	Avg. Speedup if better	1.16	1.15	1.25	1.25
	Max Speedup	1.25	1.21	1.99	1.48
`CSRbyNZ`	# matrices is better	5	3	7	8
	# matrices is worse	0	0	0	0
	Avg. Speedup	1.03	1.02	1.04	1.04
	Avg. Speedup if better	1.15	1.17	1.16	1.16
	Max Speedup	1.19	1.18	1.26	1.26
`Unfolding`	# matrices is better	5	4	7	6
	# matrices is worse	0	0	0	1
	Avg. Speedup	1.01	1.00	1.03	1.02
	Avg. Speedup if better	1.11	1.13	1.28	1.28
	Max Speedup	1.12	1.19	1.67	1.60

Table 3.3: Speedup obtained by using loop Distribution and randomization.

race condition, as described in Section 3.3.

Table 3.3 shows the performance improvement by using loop distribution and randomization. To compute the numbers on this table, we only take into account differences in running times of 10% or more. This guarantees that the numbers reported in the table are the result of our strategy and not due to different running times across different executions. For each machine and method, the table shows the number of matrices that have a performance improvement over 10%, the number of matrices that have a performance drop of 10%, the average overall speedup, the average speedup of that method only when it has performance improvement, and the maximum speedup of that method.

As the table, shows loop distribution and randomization can reduce the load imbalance, reducing execution time in most cases. In the best case, it obtains an speedup of 1.98x (debr matrix with `stencil` and running on i2pc3). In one case, this strategy results on a performance drop of 18% (s3dkq4m2 matrix with `Unfolding` and running on i2pc5). For many matrices, this strategy has no impact, as the matrix does not suffer from load imbalance.

3.5 Latency

In this work, we are not considering issues of latency, so our remarks here will be very brief. Note that latency comes from the need to re-order data and the need to generate code. `CSR` and $\texttt{CSR}_u$ do neither, and have no latency; all other methods do code generation.

`CSRbyNZ`, `Stencil` and `GenOSKI` all involve some kind of analysis prior to code generation: grouping the rows by non-zero count, calculating the stencil of each row, classifying blocks by pattern. In general, we have found that low-level code generation is the most expensive part of the specialization process, and therefore code size is the most reliable guide to specialization cost. Size was discussed when presenting the methods: in practice, `Unfolding` produces the longest code, `CSRbyNZ` almost always produces code of modest size (though much bigger than `CSR`), while the amount of code produced by `Stencil` and `GenOSKI` varies by matrix. (We note that when those two methods do produce large codes, they usually do not perform very well.) Performance issues, and their relation to code size, are discussed further in Section 5.

3.6 Discussion

We would like to mention two other potentially useful methods which we are not testing in this study, *vector instructions* and *mixed methods*. In general, our methods cannot efficiently use vector units, due to non-consecutive accesses of vector `v`. For matrices that are almost perfectly banded, elements can be stored in diagonal form, and vector units can be used to advantage. However, in our experiments with this method, it was never the best for our set of matrices. Similarly, regular (non-generative) OSKI never showed well for us. Thus, we do not show results for these two methods.

Another option is to use mixed methods, where a matrix is decomposed into two or more matrices, and each matrix is handled with a different method. For example, we might use the `Stencil` method for the dense bands around the diagonal and `CSRbyNZ` for the remaining elements. We have experimented with this idea, but we have only rarely seen it perform well. Furthermore, the algorithmic space here is so large that it is not yet clear to us how to go about exploring it. For both these reasons, we do not show results for mixed methods here.

Chapter 4

Experimental Setup

We have implemented and evaluated the following methods: `CSR`, $\texttt{CSR}_u$ with u ranging from 1 to 3, `CSRbyNZ`, `Unfolding`, `Stencil`, and `GenOSKI`. In our experiments, `CSR` performs a bit better than $\texttt{CSR}_u$ in most of the cases so that we only report $\texttt{CSR}_1$ results. For `GenOSKI` we only report results for split-by-count. For `CSRbyNZ` and `Stencil`, we report results for both, split-by-pattern and split-by-count. With the split-by-pattern approach, when a loop has to handle more than $nthread \times 500$ non-zeros, we split the loop to allow for a better balanced workload. For `GenOSKI`, our experiments show that the best results are obtained with blocks of 4×4 or 5×5, so we only show results for these sizes, and use the names `GenOSKI4` and `GenOSKI5`, respectively.

We compare our methods against the Intel MKL library version 14.0 using four threads. The four target platforms on which we ran our experiments are listed in Table 4.1. To generate parallel code we used the OpenMP "section" construct and created as many sections as threads. The codes were compiled with icc with `-O3 -openmp` compiler flags.

Name	Processor & Freq (GHz)	Cores (SMP cores)	Cache Sizes (Bytes)			Mem	OS	icc
			L1 (I/D)	L2	L3	(GB)		
loome2	Intel Core i7 880 @ 3.07	4 (8)	32K	256K	8M	8	Linux CentOS 5.8	14.0.2.144
loome3	Intel Core i5 2400 @ 3.10	4 (4)	32K	256K	6M	8	Linux CentOS 5.8	14.0.2.144
i2pc3	Intel Xeon E7-4860 @ 2.27	10 (80)	32K	256K	24M	128	Scientific Linux 6.3	14.0.2.144
i2pc5	Intel Xeon L7555 @ 1.87	8 (64)	32K	256K	24M	64	Scientific Linux 6.3	14.0.2.144

Table 4.1: Specification of experimental machines.

Table A.1 shows the 88 matrices we use. They were obtained from the Matrix Market [9], the University of Florida Sparse Matrix collection [10, 11], or from the discretization of a Poisson problem and used as GPU SpMV data sets [12, 13]. Many of them have over millions of none-zero elements. This helps us understand the scalability of our specialization methods with large matrices. The table is sorted by number of non-zeros. Some matrices are derived from graphs that model social or communication networks following a power law distribution, while others come from Finite Element modeling (SPARSKIT), etc. Several of these matrices have been used in previous studies [17, 15, 21, 13]. We did not select them based on any specific pattern, but rather to have matrices that represent a variety of domains. Table A.1 also provides the following information: the name and group of the matrices. Group "MM" stands for Matrix Market, and "SpGEMM" stands for the matrices obtained from the discretization of a Poisson problem [12, 13]; "FL" stands for Florida Sparse Matrix collection, while the name after "'FL:", e.g. SNAP, shows the group of the matrix; p indicates whether the matrix is a pattern matrix. Notice that some of these matrices are *pattern matrices*, for which the source does not provide values; we have generated values for these matrices, with all the generated values being different.

Table A.2 provides the matrix characteristics: n and nnz; the denseness (nnz/n); The last few columns give data that are useful in evaluating the performance of these methods: *stencils* is the number of different stencils; *genOSKI4* and *genOSKI5* are the numbers of distinct patterns that appear in 4×4 and 5×5 blocks, respectively; *distVals* is the number of distinct values; and *Row_nz* is the number of distinct row non-zero counts; *emptyrow* is the number of rows that have no non-zero elements.

Table A.3 shows code and data size for the matrices for the different methods when we generate OpenMP code for 4 threads. These sizes are drawn directly from the compiled code. Code size values differ slightly from those computed using the expressions in Table 3.1, as those expressions do not take into account the extra loops that appear when a loop is split for parallel execution into 2 or more threads. Also, the icc compiler unrolls some loops. In addition, to speed up compilation time[1], we split the code into several functions, grouped in multiple files. As a consequence, even if a matrix has a single distinct value, this value will appear once in each file. Thus, for `Unfolding`, the data size in practice is larger than the number of distinct values reported in the table.

To collect the timings, we did the following for each matrix/method/machine combination: (1) Performed matrix-vector multiplication 10,000 times (on an unloaded machine); (2) repeated (1) five times; and (3) chose the fastest of those five trials. Before each call to the multiplication function, the output vector is zeroed.

We also compare our methods against two state-of-the-art SpMV libraries, `BiCSB` [14] and `CSX` [15], that have online code that can be installed and run. `BiCSB` [16] is implemented on top of CSB [17], a new parallel sparse matrix data structure that allows efficient SpMV on multicores. `BiCSB` uses bitmasked register blocks to reduce the memory bandwidth requirement when using register blocking[2]. `CSX` [18] is based on the Compressed Sparse eXtended (CSX) format that allows for a flexible storage format to support a variety of structures within the sparse matrix, such as horizontal, vertical, diagonal, antidiagonal, or blocks. This approach requires runtime code generation. We compare against the SpMV running times, without taking

[1] Compilers have been optimized to compile code written by humans, which tends to be small, and so they are slow when compiling large codes produced with a code generator, as we do.

[2] We ran both CSB and `BiCSB`, but since `BiCSB` is always faster than CSB we only compare against `BiCSB`.

email-EuAll	cit-HepPh	soc-Epinions1	soc-sign-Slashdot081106	web-NotreDame
webbase-1M	e40r5000	fidapm11	fidapm37	m133-b3
torso2	fidap011	cfd2	m14b	s3dkt3m2
conf6_0-8x8-20	ship_003	cage12	debr	mc2depi
s3dkq4m2	engine	thermomech_dK		

Table 4.2: 23 matrices used for `BiCSB` and `CSX`

into consideration the time to generate the code. For `CSX`, we encountered library conflicts on i2pc3 and i2pc5 and input format issues for many matrices. Thus, we select 23 matrices, listed in Table 4.2, to run `BiCSB` on all four machines and `CSX` on loome2 and loome3.

Chapter 5

Experimental Results

In this section, we report our experimental results. We compare the running times of our methods in detail with MKL for all 88 matrices and four machines using four threads. We discuss how the characteristics of the machines and matrices help explain the timing results; the latter is important in the process of predicting the best method. We briefly address the issue of scalability by comparing our methods to Intel MKL library when running on eight threads (rather than our usual four). Finally, we evaluate two state-of-the-art libraries, CSX and BiCSB, comparing to our methods and MKL library for 23 out of 88 matrices.

5.1 Comparison of Methods

Table A.4, A.5, A.6 and A.7 show, for all 88 matrices and four machines, the speedup of MKL, CSR, Stencil, GenOSKI4, GenOSKI5, Unfolding, CSRbyNZ with respect to MKL, where the speedup is computed by dividing the MKL running times by the running times of each method, when all run with four threads (including MKL). The table also shows the best method for each matrix. Of course, the best method is MKL if the "BestMethodSpeedup" is below one. The last row of each table shows the average values of each method and the best method. For CSRbyNZ and Stencil we compare against the code version, split-by-pattern or split-by-count, that performs the best. For GenOSKI we only compare agains split-by-count. Table A.8 compares the performance of split-by-pattern and split-by-count for CSRbyNZ and Stencil for the different machines and matrices. Running times are similar, although split-by-pattern is usually faster, but not always.

Table 5.1 compares the different methods. For each method and machine the table shows the average speedup if that method is used for all the matrices, the number of matrices for which that method is the best, the number of matrices that run faster than MKL using that method, and the average speedup of that method if only used when it runs faster than MKL. The last two metrics tell us how often each method improves with respect to MKL, and if it improves, what is the average speedup. The last row in the table (labeled Best) shows the same metrics, but when the best specializer is chosen. In this case, "Avg. speedup" is the speedup

		loome2	loome3	i2pc3	i2pc5
CSR	Avg. Speedup	1.090	1.117	1.100	1.068
	# matrices is best	19	22	26	34
	# matrices is better	82	81	71	78
	Avg. Speedup if better	1.099	1.132	1.138	1.103
Stencil	Avg. Speedup	1.036	0.952	1.102	1.020
	# matrices is best	20	11	12	7
	# matrices is better	45	38	53	41
	Avg. Speedup if better	1.404	1.346	1.334	1.334
GenOSKI4	Avg. Speedup	1.055	1.052	1.025	0.975
	# matrices is best	9	13	5	4
	# matrices is better	53	50	45	37
	Avg. Speedup if better	1.243	1.269	1.196	1.178
GenOSKI5	Avg. Speedup	0.974	0.952	0.988	0.929
	# matrices is best	2	3	1	1
	# matrices is better	42	37	41	29
	Avg. Speedup if better	1.248	1.274	1.170	1.183
Unfolding	Avg. Speedup	1.008	0.899	1.263	1.181
	# matrices is best	16	14	33	35
	# matrices is better	32	26	48	43
	Avg. Speedup if better	1.740	1.688	1.738	1.720
CSRbyNZ	Avg. Speedup	1.162	1.189	1.077	1.001
	# matrices is best	22	25	8	6
	# matrices is better	61	68	43	40
	Avg. Speedup if better	1.271	1.278	1.285	1.208
Best specialization	Avg. speedup	1.453	1.437	1.470	1.416
	#matrices is better	88	88	85	87
	Avg. Speedup if better	1.453	1.437	1.488	1.421

Table 5.1: Comparison between methods.

obtained if we always use a method that requires specialization (in some cases that will result in slowdowns with respect to MKL). Notice that this value is very similar to the Avg. speedup of the best method, shown in last row of Table 5.1.

Overall, the results show that specialization can produce significant speedups. Out of 88 matrices, specialization produces speedups for 88, 88, 85, and 87 matrices and average speedups of 1.453, 1.437, 1.470, and 1.416 for loome2, loome3, i2pc3, and i2pc5, respectively. The average speedups are computed using the best method using specialization, even if this method is slower than a method that does not require specialization.

5.2 Explaining the Timings

The natural question is how to determine what is the best method. Our results show that speedups depend on both machine and matrix characteristics. For many matrices, 38 out 88 matrices, listed in Table 5.2, the same method is the best across the board. For many others, the best method varies across machines. For instance, for email-euAll and cage12, there are four different methods with very different speedups. We

Matrix	Method	n	nnz	nnz/n	stencil	genOSKI4	genOSKI5	distVals	rowNZ	E.row	minS	maxS
minnesota	Unfolding	2642	3303	1.25	551	211	441	3303	3	164	1.35	1.97
pde900	Unfolding	900	4380	4.86	9	6	4	3248	3	0	1.41	1.94
dw2048	Unfolding	2048	10114	4.93	18	8	29	693	5	0	1.58	1.80
orsreg_1	Unfolding	2205	14133	6.40	27	17	21	111	4	0	1.67	2.25
mcfe	CSR	765	24382	31.87	346	391	689	24381	55	0	1.30	1.42
fidap002	CSR	441	26831	60.84	436	93	112	11118	22	0	1.11	1.21
cavity05	CSR	1182	32632	27.60	395	181	310	3280	30	0	1.09	1.27
bcsstk13	CSR	2003	42943	21.43	1820	1284	2241	13781	73	0	1.06	1.26
fidap024	CSR	2283	47897	20.97	622	339	552	20387	26	0	1.04	1.15
fidap010	CSR	2410	54816	22.74	356	188	318	22939	27	0	1.08	1.16
cavity15	CSR	2597	71601	27.57	371	183	276	48418	26	0	1.09	1.20
fidap013	CSR	2568	75628	29.45	1264	225	433	39097	22	0	1.04	1.14
utm5940	genOSKI5	5940	83842	14.11	176	162	47	82768	25	0	1.10	1.35
fidap031	CSR	3909	91165	23.32	745	402	694	35726	39	0	1.03	1.11
memplus	CSRbyNZ	17758	99147	5.58	16719	605	1354	50039	91	0	1.52	1.93
as-caida	Unfolding	31379	106762	3.40	25184	371	755	4	158	4904	2.05	2.50
cavity23	CSR	4562	131735	28.87	440	170	293	90994	26	0	1.07	1.11
bcsstk16	CSR	4884	147631	30.22	301	246	404	15779	40	0	1.02	1.15
usroads	CSRbyNZ	129164	165435	1.28	21157	688	1893	165435	4	6173	1.44	2.14
chem_master1	Unfolding	40401	201201	4.98	9	9	10	20801	3	0	1.64	2.31
enron	CSR	69244	276143	3.98	12725	5191	9578	276143	370	51676	1.16	1.28
af23560	genOSKI4	23560	460598	19.55	122	3	98	310480	12	0	1.18	2.56
soc-sign-Sla.	Unfolding	77357	516575	6.67	40649	1212	2867	2	279	34008	2.58	2.93
m133-b3	Unfolding	200200	800800	4	200200	489	1627	2	1	0	1.19	1.40
s3dkt3m2	Stencil	90449	1888336	20.87	935	97	143	29116	23	0	1.15	1.63
cant	Stencil	62451	2034917	32.58	90	182	288	108	36	0	1.14	1.63
mc2depi	Unfolding	525825	2100225	3.99	2298	50	57	3584	3	0	1.24	1.66
engine	Unfolding	143571	2424822	16.88	84195	108	538	1	147	0	2.86	3.89
apache2	Unfolding	715176	2766523	3.86	10	10	19	41	4	0	1.61	1.89
thermomech_dK	genOSKI4	204316	2846228	13.93	204290	17	329	1967432	9	0	1.00	1.11
webbase-1M	Unfolding	1000005	3105536	3.10	504865	4394	11141	222	370	0	1.33	1.74
amazon0601	CSRbyNZ	403394	3387388	8.39	401861	11089	30204	3387389	10	955	1.02	1.76
sqr_mtx_aniso.	CSRbyNZ	832081	5797879	6.96	828753	2416	8402	4361273	8	0	.093	1.15
pwtk	Stencil	217918	5871175	26.94	9183	662	1214	5592868	78	0	1.13	1.69
horseshoe_mtx	CSRbyNZ	853761	5947651	6.96	850178	349	1501	4558272	6	0	1.01	1.16
atmosmodj	Unfolding	1270432	8814880	6.93	27	4	28	5	4	0	2.05	4.42
struct._2d_9pt	Unfolding	1048576	9424900	8.98	9	3	28	3	3	0	2.90	5.06
mesh_3d_h015	CSR	1088958	15392990	14.13	967799	41773	286900	8119845	37	0	1.03	1.06

Table 5.2: Matrices where the same method is the best on all platforms.

now discuss how the machine and matrix characteristics (Tables 4.1, A.2, A.3) help explain the timings (Tables 5.1).

5.2.1 Unfolding

`Unfolding` is the best method when the sum of code and data size fits in the Last Level Cache (LLC) (Table 4.1 and A.3). Many of our matrices are large, and should be large for those matrices. However, as explained in Section 2, when the number of distinct values is small (*distVals* in Table A.2), the compiler can apply certain optimizations such as CSE, that significantly reduce the code size.

From Table 5.2, we see that the matrices that benefit from this method are: minnesota, ped900, dw2049, orsreg_1, as-caida, chem_master1, soc-sign, m133-b3, mc2depi, engine, apache2, webbase-1M, atmosmodj and structured_2d_9pt. Among these matrices, minnesota, ped900, dw2049, and orsreg_1 are small matrices. `Unfolding` is the best method for these four matrices because all of them can fit into the cache (Table A.3) and `unfolding` uses the least number of instructions, as discussed in Section 2.4.

Matrices as-caida, soc-sign, m133-b3, engine, apache2, atmosmodj and structured_2d_9pt have only 4, 2, 2, 1, 41, 5 and 3 distinct values, respectively, and achieve very good speedups in all the platforms. m133-b33 obtains, in general, lower speedups than soc-sign and engine, even though it only has 2 distinct values. The reason is that the code size of `unfolding` for m133-b33 is about the size of the `CSR` data.

For webbase-1M, the number of distinct values is 222, but it is a large matrix in terms of non-zero elements, and thus `unfolding` is the best for all machines.

The matrix chem_master1 has 20801 distinct values, with very few stencils, genOSKI and CSRbyNZ patterns (9, 9, 10, and 3, respectively). However, `unfolding` still achieves good performance because `unfolding` can take advantage of the relatively small number of non-zeros, reducing the bandwidth requirements. The results also show that for i2pc3 and i2pc5, `unfolding` is the best method for 33 and 35 matrices. This is because both machines have the largest LLC (24MB).

To the best of our knowledge, this is the first study that reports the benefit of `Unfolding` when the number of distinct values is small. This can be applicable to a large set of matrices, like those derived from graphs, such as the adjacency matrix or laplacian matrix. Another example are algebraic multi-grid methods for sparse linear systems [22].

5.2.2 Stencil

`Stencil` has the potential to produce good speedups, but only the matrices with a small number of stencils can benefit from it. `Stencil` is the best method on all platforms for s3dkt3m2, cant, and pwtk which have 935, 90 and 9183 stencils, respectively. The number of stencils by itself may not be enough to determine that `stencil` is the best method comparing to other properties. s3dkt3m2 has only 23 CSRbyNZ patterns. cant has 108 distinct values and 36 CSRbyNZ patterns. pwtk has 662 stencils and 78 CSRbyNZ patterns. However, for these matrices `stencil` produces the least code and data size for these matrices (see table A.3).

`Stencil` is also usually good for torso2, m2depi, and e40r500. Although for these matrices `Stencil` is not the best for all the machines, `Stencil` is usually almost as good as the best. Notice that these matrices (together with conf6_0-8x8-20) are the matrices with the smallest number of stencils. This method delivers significant speedups, when it is better than `MKL`, as shown in Table 5.1.

5.2.3 CSRbyNZ

`CSRbyNZ` always produces small codes. Even for the power law matrices (matrices from the SNAP group and webbase-1M) that have a relatively large *Row_nz* (see Table A.2), it is still much smaller than the number of stencils or block patterns. This method tends to have modest speedups. The code executes fewer loop overhead instructions, resulting in higher Instruction Level Parallelism (ILP). The data size of this method is similar to that of CSR, but it requires less data size when the matrix has a higher number of empty rows, because only the rows that have non-zero elements are relevant in this method.

CSRbyNZ is the best method for memplus, usroads, amazon0601, square_matrix_anisotropic and horseshoe_matrix_anisotropic, which have 91, 4, 10, 8 and 6 CSRbyNZ patterns, respectively. It is the best method for memplus because it has small number of CSRbyNZ patterns while its other properties such as stencils, genOSKI patterns and distinct values are not so good. For the other three matrices, the small number of CSRbyNZ patterns already show its superiority.

We consider this to be a default method that can be used when none of the other methods seems appropriate. It is interesting to notice that many of the power law matrices benefit from this method in loome2, and loome3 machines, which have smaller caches than i2pc3 and i2pc5.

5.2.4 GenOSKI

GenOSKI always produces modest code size (see Table A.3), as the number of patterns is never too big: out of 65,535 possible patterns when using blocks of size 4×4, the maximum in Table A.2 is 4,394 (if we discount mesh_3d_h015 and amazon0601). The ability of this method to decrease data size is also important, and that depends on the number of blocks that are empty (each block needs a `cols` and a `rows` index) and the locality.

The matrices utm5940, af23560, thermomech_dK are the ones for which genOSKI is the best method across machines. genOSKI4 produces significant speedup for af23560 and thermomech_dK, because they only have 3 and 17 genOSKI4 patterns, respectively, resulting in smaller codes. utm5940 profits from genOSKI5 because this method produces the smallest code and data.

Speedups of this method are comparable to those of CSRbyNZ. loome2 and loome3 stand out as the machine most favorable to GenOSKI. (4×4 is usually the best block size; 5×5 is occasionally better. We have also evaluated smaller blocks, but we do not report results, as they are never better.)

5.2.5 CSR

CSR is the very basic and simple method to perform sparse matrix-dense vector multiplication. It is the best method in some cases where the density (nnz/n in Table A.2 and 5.2) is high. CSR uses two nested for loops to iterate each non-zero elements. If the density is high, then there are more non-zeros in a row, and as a result the overhead of the outer loop can be amortized.

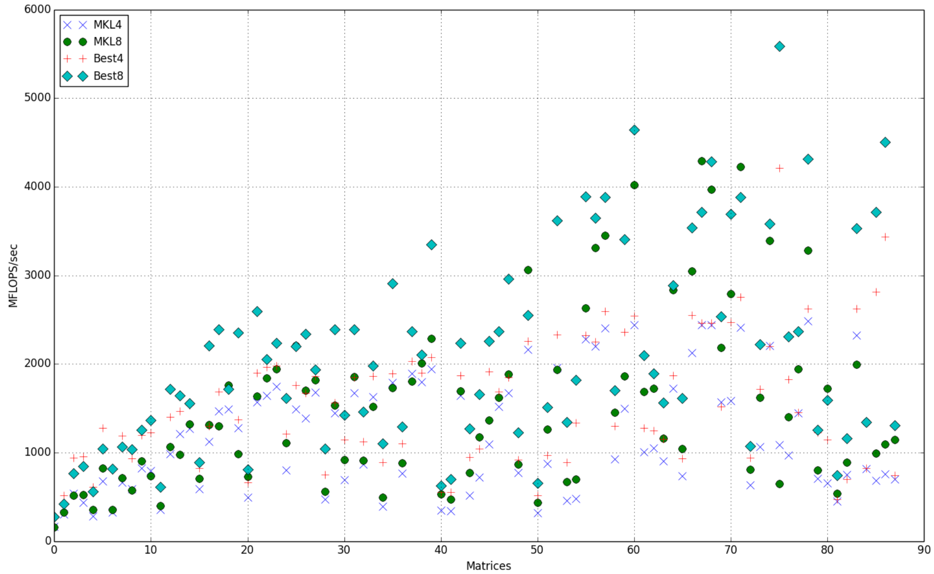

Figure 5.1: MFLOP/sec for MKL and Best Method with 4 and 8 threads on i2pc3.

5.2.6 MKL

MKL, the baseline Intel library, is usually not the best method in our experiments. MKL is the best only for 1 matrix on loome3, 3 matrices on i2pc3 and 1 on i2pc5.

5.3 More Parallelism

We have also run the experiments with eight threads on i2pc3 and i2pc5 to evaluate the scalability of our methods. Figure 5.1 and 5.2 show the throughput (MFLOPS/sec) for MKL and the best of our methods with 4 and 8 threads on i2pc3 and i2pc5. The figures show that in most cases, a method that requires code generation performs better than MKL. When the matrix size is small, the multi-threading overhead can be high, in which case the runs with fewer threads take less time and obtain a higher throughput. Moreover, we see the best specialized method with 8 threads has significantly better performance than any other configurations, meaning that the methods that require specialization scale better than MKL.

5.4 Comparison with State of the Art Libraries

Other than Intel MKL library, we also have compared our methods with BiCSB and CSX. We could not run all the matrices with BiCSB due to some matrix format issue that we could not address. i2pc3 and i2pc5 also

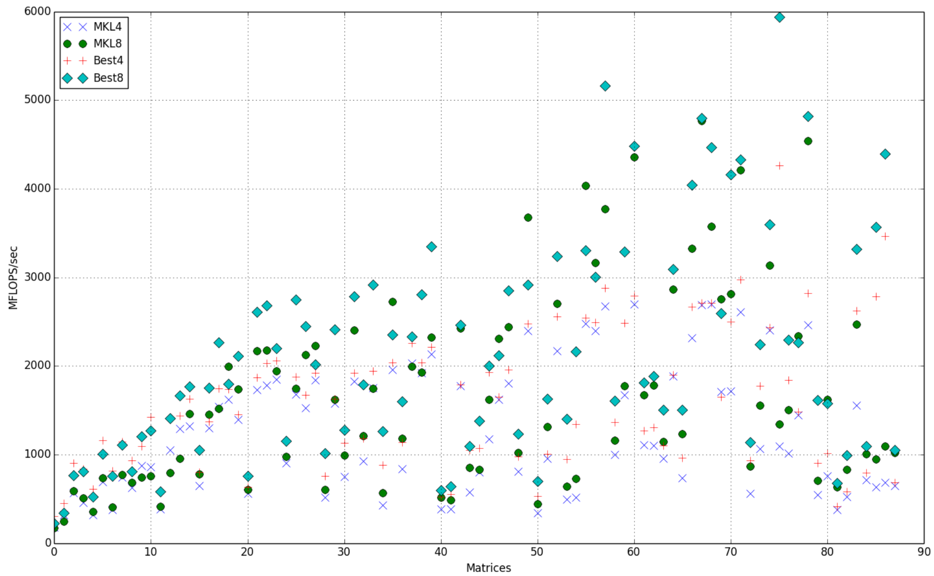

Figure 5.2: MFLOP/sec for MKL and Best Method with 4 and 8 threads on i2pc5.

have library conflicts (Boost library). Thus, we select 23 matrices to run `BiCSB` on all the machines and `CSX` on loome2 and loome3.

Figures 5.3, 5.4, 5.5, and 5.6 show MFLOPS/sec for all four machines for the selected 23 matrices for `MKL`, `Best Specializer`, `BiCSB` and `CSX`. `Best Specializer` is the best method among `CSR`, `stencil`, `genOSKI4`, `genOSKI5`, `CSRbyNZ`, and `unfolding`. Notice that `CSX` is not shown for i2pc3 and i2pc5, as dicussed above. Results in these figures show that `Best Specializer` is usually also faster than `MKL`, `CSX` and `BiCSB`. Moreover, i2pc3 has a similar behavior as i2pc5 and loome2 is similar to loome3.

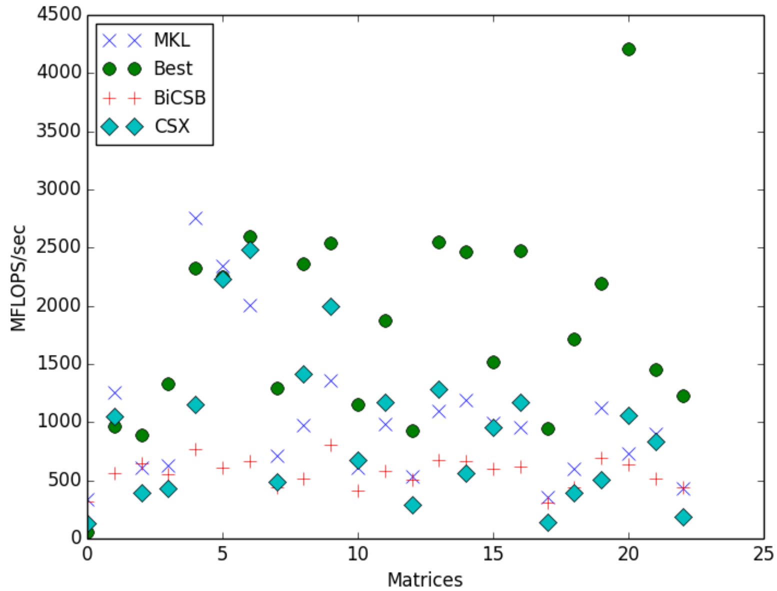

Figure 5.3: MFLOPs/sec for loome2 for Best Specializer, MKL, BiCSB, and CSX.

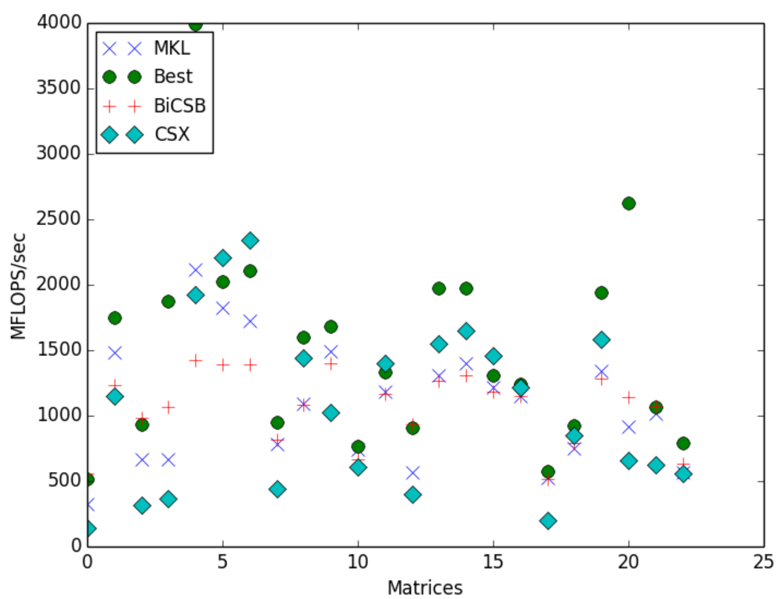

Figure 5.4: MFLOPs/sec for loome3 for Best Specializer, MKL, BiCSB, and CSX.

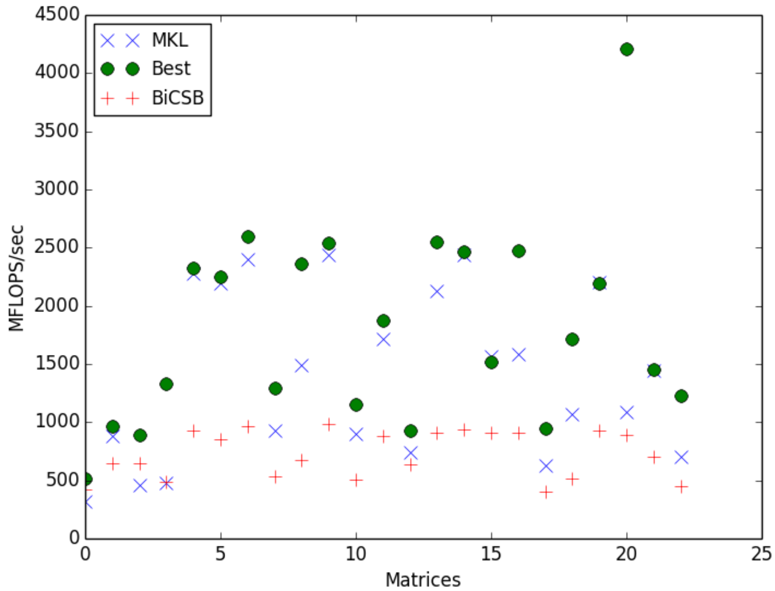

Figure 5.5: MFLOPs/sec for i2pc3 for Best Specializer, MKL, and BiCSB.

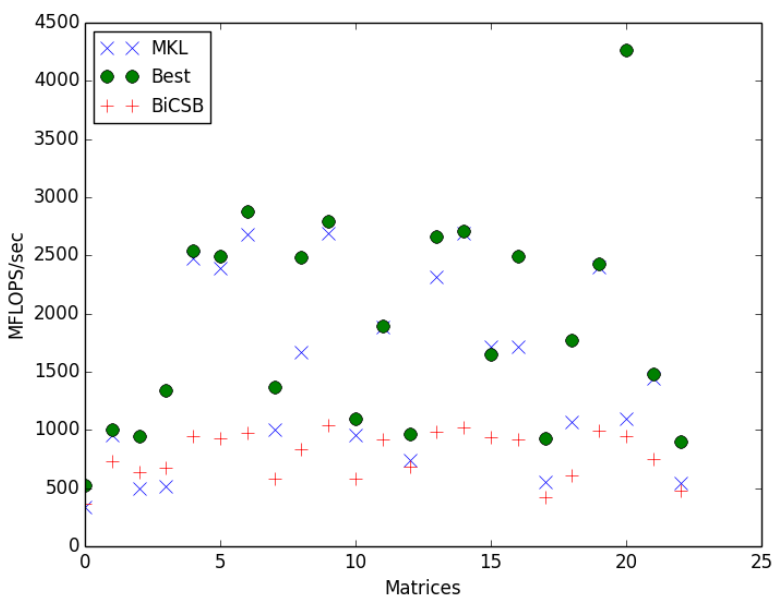

Figure 5.6: MFLOPs/sec for i2pc5 for Best Specializer, MKL, and BiCSB.

Chapter 6

Applications

Knowing that efficient codes can be produced by code generation is interesting, but is it useful? That depends entirely upon the tolerance for latency in the particular application.

We note that it is very common for the *shape* of a matrix — the exact locations of its non-zeros — to be known even when the values are not. Some of these are referred to as "pattern matrices," and the Matrix Market and the Florida collection include many of them. Also, for those matrices derived using Finite Element methods [23], the shape of the matrix is usually known ahead of time, as the matrix is derived from a mesh that is usually available before solving the problem. All of our methods except unfolding generate code based only on the shape; by generating code for those matrices off-line — only the `mvalues` array needs to be supplied at runtime — the issue of latency is entirely obviated.

The more challenging case is when nothing is known about the matrix until runtime. The work presented here is the first step in the creation of a library for matrix-vector multiplication that will use run-time specialization, *auto-tuning*, and *machine learning techniques* to predict the best method, as has been done in previous work [24, 25, 26, 27, 19]. The library would be employed in cases where a single matrix M is to be multiplied by many vectors. Here is how we envision the library working.

The user will supply the matrix to the library, and the library will produce a pointer to a function of type `void multByM (double v[], double w[])`. When called subsequently, `multByM` will multiply M by `v` and place the result in `w`. (The OSKI library [8, 19, 20] operates similarly.)

When first presented with M, the system will determine which method will produce the most efficient `multByM`. It may determine that `CSR` is the best, and will immediately return a pointer to pre-existing code; or it may determine that a specialized code, which must be generated at runtime, will be most efficient. This process itself will take time, and generating the specialized code, if that is the decision, will take even more; in any case, the system cannot produce overall speed-ups if the matrix is to be multiplied only a small number of times. (The risk might be managed by running program generation in parallel with a low-latency method like `CSR` until the generated code is ready.)

This library organization raises several questions:

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

2. How can the system determine the best method for a particular matrix on a particular machine?

3. How can the latency introduced by the code specialization process be minimized?

Question (2) will be addressed by auto-tuning [24, 25, 26, 27, 19]. Here, one gathers information about the machine at "install time," and feeds it into the runtime specialization process, which uses it, together with characteristics of the matrix M, to determine how best to generate `multByM`.

To minimize latency (question 3), we are developing specialized code generators for this problem.

Chapter 7

Related Work

Sparse matrix-dense vector multiplication is an operation that is used in many scientific problems. It has been studied in the OSKI project [8]. A number of researchers have looked at multi-core implementations [28, 29, 30, 17, 14, 15, 21, 7, 6]. Among those, we have compared our codes with `CSB`, `BiCSB` [14], and `CSX` [15], as their libraries were available on line. `CSRByNZ` is similar to the method described by Mellor-Crummey and Garvin [6], while `GenOSKI` is similar to PBR [7]. Perhaps, the main difference between our work and previous ones, is that rather than evaluating a single method, we are evaluating many. Our goal was to understand if, and by how much, specialization could improve performance.

As discussed in Section 6, auto-tuning is used to overcome the problem that the best code for a problem can vary from machine to machine. It is used by OSKI; other examples are [24, 25, 26, 27].

To improve performance, languages like Java, Javascript or Python have a just-in-time compiler to generate more efficient code. Compilers like the Google V8 compiler for JavaScript [31], where the programmer does not declare the type of the variables, specialize the code to the variable that appears more often. Our approach is similar to this in that we are specializing to the data of the matrix M that repeats, rather than the type of the variable. It differs in that we know the algorithm that is being generated, and as a result we can do more optimizations. Runtime specialization is a new optimization technique, and it is important to evaluate in real codes what is the performance benefit that can be obtained, and what is the performance degradation that can be suffered.

The area of program specialization — also called *code generation*, *partial evaluation*, or *staging* — has been quite heavily studied, especially with respect to language features, such as type-checking, that promote simplicity and safety of specialization [1, 2, 3]. Program specialization using explicit annotations has received extensive attention. However, much of the research has focused on language infrastructure, especially on type systems to statically guarantee safety of the generated program. Examples in those papers tend to be small-sized, with no or very little benchmarking results. In a recent Nii Shonan Meeting, the potential of using program specialization on high performance computing problems was identified [5]. A set of problems is given as "Shonan Challenge" – a list of HPC problems amenable to specialization, among which there is sparse-matrix algebra library as well. Program specialization is used to address a realistic problem, Gaussian Elimination [32], where a highly configurable generator is written that is able to produce numerous different versions of the algorithm based on parameters such as matrix representation, pivoting policies and result types. Program generation has been shown to produce faster marshalling (a.k.a. serialization) routines for particular data types by specializing the program at run-time [33] or by benefitting from the statically available information [34]. Work in this area specifically addressing high-performance for realistic applications includes work on marshaling [33, 34] and on code-optimizing transformations [35]. The transformations include loop unrolling, tiling, pipelining, scalar promotion, etc. It is shown that competitive performance can be obtained using the generative approach. In none of these, autotuning is proposed to select the best specializer out of several candidates – to our knowledge, ours is the first library to consider the combination of autotuning and specialization. With *runtime* specialization, the focus moves toward the efficiency of specialization itself [36, 37].

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

Chapter 8

Conclusions

In this thesis we have shown that specialization can be used to obtain speed-ups for SpMV. Our experimental results using 88 matrices and four machines show that a method requiring specialization runs faster than a method without specialization in 347 out of 352 trials (88×4). These experimental results include comparisons with state of the art libraries, such as Intel's MKL, BiCSB, and CSX. If we only use specialization, the average speedup with respect to Intel's MKL library ranges from 1.41x to 1.47x, depending on the machine. For individual matrices, these speedups can be higher.

In this thesis, rather than evaluating a single method, we are evaluating many. Our results show that there is no one best method and that the best method depends on the machine and matrix characteristics. Among the evaluated methods, we have found that one of our methods, `Unfolding`, can produce significant speedups when the number of distinct values is small. This is important, as this can be common in matrices that are derived from graphs, such as the Laplacian matrix, or algebraic multigrid methods for sparse linear systems.

Appendix A

Table A.1: List of matrices used in the experiments.

ID	Name	group	p	ID	Name	group	p
1	minnesota	FL:Gleich	Y	45	ca-AstroPh	FL:SNAP	Y
2	pde900	MM	N	46	chemmaster1	FL:Watson	N
3	dw2048	MM	N	47	fidap035	MM	N
4	add20	MM	N	48	bcsstk17	MM	N
5	as-735	FL:SNAP	Y	49	enron	FL:LAW	Y
6	orsreg1	MM	N	50	e30r0500	MM	N
7	ca-GrQc	FL:SNAP	Y	51	email-EuAll	FL:SNAP	Y
8	bcsstk26	MM	N	52	cit-HepPh	FL:SNAP	Y
9	add32	MM	N	53	af23560	MM	N
10	sherman5	MM	N	54	soc-Epinions1	FL:SNAP	Y
11	saylr4	MM	N	55	soc-sign-Slashdot-081106	FL:SNAP	N
12	Oregon-1	MM	Y	56	e40r5000	MM	N
13	mcfe	MM	N	57	fidapm11	MM	N
14	fidap002	MM	N	58	fidapm37	MM	N
15	cavity05	MM	N	59	m133-b3	FL:JGDHomology	N
16	p2p-Gnutella04	FL:SNAP	Y	60	torso2	FL:Norris	N
17	bcsstk13	MM	N	61	fidap011	MM	N
18	fidap024	MM	N	62	maceconfwd500	FL:Williams	N
19	fidap010	MM	N	63	cop20kA	FL:Williams	N
20	bcsstk15	MM	N	64	web-NotreDame	FL:SNAP	Y
21	p2p-Gnutella24	FL:SNAP	Y	65	cfd2	FL:Rothberg	N
22	mhd3200a	MM	N	66	m14b	FL:DIMACS10	Y
23	cavity15	MM	N	67	s3dkt3m2	MM	N

Table A.1 continued: List of matrices used in the experiments.

ID	Name	group	p	ID	Name	group	p
24	fidap013	MM	N	68	conf60-8x8-20	FL:QCD	N
25	bcsstk18	MM	N	69	qcd54	MM	Y
26	bcsstk24	MM	N	70	ship003	FL:DNVS	N
27	utm5940	MM	N	72	cage12	FL:vanHeukelum	N
28	fidap031	MM	N	72	cant	FL:SNAP	N
29	ca-CondMat	FL:SNAP	Y	73	debr	FL:AG-Monien	Y
30	fidap015	MM	N	74	mc2depi	FL:Williams	N
31	memplus	MM	N	75	s3dkq4m2	MM	N
32	mhd4800a	MM	N	76	engine	FL:TKK	N
33	wiki-Vote	FL:SNAP	Y	77	apache2	FL:GHSpsdef	N
34	s3rmt3m3	MM	N	78	thermomech-dK	FL:Botonakis	N
35	as-caida	FL:SNAP	N	79	consph	FL:Williams	N
36	bcsstk28	MM	N	80	webbase-1M	FL:Williams	N
37	ca-HepPh	FL:SNAP	Y	81	amazon0601	FL:SNAP	Y
38	cavity23	MM	N	82	web-Google	FL:SNAP	Y
39	s2rmq4m1	MM	N	83	squarematrixanisotropic	SpGEMM	N
40	bcsstk16	MM	N	84	pwtk	FL:Boeing	N
41	usroads-48	FL:Gleich	Y	85	horseshoematrixanisotropic	SpGEMM	N
42	usroads	FL:Gleich	Y	86	atmosmodj	FL:Bourchtein	N
43	fidapm29	MM	N	87	structured2d9pt	SpGEMM	N
44	email-Enron	FL:SNAP	Y	88	mesh3d-h015	SpGEMM	N

Table A.2: Characteristics of the matrices used in the experiments.

ID	n	nnz	nnz/n	stencil	genOSKI4	genOSKI5	distVal	rowNZ	emptyrows
1	2642	3303	1.25	551	211	441	3303	3	164
2	900	4380	4.87	9	6	4	3248	3	0
3	2048	10114	4.94	18	8	29	693	5	0
4	2395	13151	5.49	2128	568	1132	7390	48	0
5	7716	13895	1.80	5470	161	352	13895	35	1756
6	2205	14133	6.41	27	17	21	111	4	0
7	5242	14496	2.77	3524	126	235	14496	48	1395
8	1922	16129	8.39	1297	310	706	13480	26	0
9	4960	19848	4.00	3941	233	364	13883	6	0
10	3312	20793	6.28	140	60	114	15096	20	0
11	3564	22316	6.26	34	18	34	11	5	0
12	11492	23409	2.04	9503	230	429	23409	47	1162
13	765	24382	31.87	346	391	689	24381	55	0
14	441	26831	60.84	436	93	112	11118	22	0
15	1182	32632	27.61	395	181	310	3280	30	0
16	10879	39994	3.68	4903	267	623	39994	37	5944
17	2003	42943	21.44	1820	1284	2241	13781	73	0
18	2283	47897	20.98	622	339	552	20387	26	0
19	2410	54816	22.75	356	188	318	22939	27	0
20	3948	60882	15.42	3314	431	1918	2218	36	0
21	26518	65369	2.47	7375	113	221	65369	43	18948
22	3200	68026	21.26	55	45	182	47873	18	0
23	2597	71601	27.57	371	183	276	48418	26	0
24	2568	75628	29.45	1264	225	433	39097	22	0
25	11948	80519	6.74	8550	1420	2873	33337	32	0
26	3562	81736	22.95	1045	118	293	58571	42	0
27	5940	83842	14.11	176	162	47	82768	25	0
28	3909	91165	23.32	745	402	694	35726	39	0
29	23133	93497	4.04	17545	209	428	93497	83	4646
30	6867	96421	14.04	73	105	134	21326	12	0
31	17758	99147	5.58	16719	605	1354	50039	91	0

Table A.2 continued: Matrix characteristics.

ID	n	nnz	nnz/n	stencil	genOSKI4	genOSKI5	distVal	rowNZ	emptyrows
32	4800	102252	21.30	55	45	179	72344	17	0
33	8297	103689	12.50	4973	800	1878	103689	237	2187
34	5357	106240	19.83	1322	117	209	100387	36	0
35	31379	106762	3.40	25184	371	755	4	158	4904
36	4410	111717	25.33	2913	140	280	110807	68	0
37	12008	118521	9.87	9207	344	706	118521	229	2352
38	4562	131735	28.88	440	170	293	90994	26	0
39	5489	134420	24.49	167	94	141	17724	29	0
40	4884	147631	30.23	301	246	404	15779	40	0
41	126146	161950	1.28	21087	663	1791	161950	4	5783
42	129164	165435	1.28	21157	688	1893	165435	4	6173
43	13668	183394	13.42	490	193	308	96959	14	0
44	36692	183831	5.01	31838	1776	3922	183831	108	1092
45	18772	198110	10.55	15650	335	704	198110	164	2631
46	40401	201201	4.98	9	9	10	20801	3	0
47	19716	217972	11.06	202	146	287	54316	17	0
48	10974	219812	20.03	6715	232	1046	117183	54	0
49	69244	276143	3.99	12725	5191	9578	276143	370	51676
50	9661	306002	31.67	476	140	253	207699	27	0
51	265214	420045	1.58	161683	499	1088	420045	311	39805
52	34546	421578	12.20	31814	315	683	421578	162	2388
53	23560	460598	19.55	122	3	98	310480	12	0
54	75888	508837	6.71	49442	3281	8439	307854	326	15547
55	77357	516575	6.68	40649	1212	2867	2	279	34008
56	17281	553562	32.03	601	130	265	368750	25	0
57	22294	617874	27.71	4682	1197	2576	88275	22	0
58	9152	765944	83.69	8391	876	2102	350166	70	0
59	200200	800800	4.00	200200	489	1627	2	1	0
60	115967	1033473	8.91	3148	81	108	806653	3	0
61	16614	1091362	65.69	7432	1684	3315	211502	71	0
62	206500	1273389	6.17	407	445	786	118307	17	0
63	121192	1362087	11.24	96936	2940	9562	955507	24	21349

Table A.2 continued: Matrix characteristics.

ID	n	nnz	nnz/n	stencil	genOSKI4	genOSKI5	distVal	rowNZ	emptyrows
64	325729	1497134	4.60	126894	4135	9474	1497134	312	187788
65	123440	1604423	13.00	46535	3422	7823	1480984	27	0
66	214765	1679018	7.82	172130	3331	9099	1679018	22	6651
67	90449	1888336	20.88	935	97	143	29116	23	0
68	49152	1916928	39.00	648	22	156	84553	1	0
69	49152	1916928	39.00	648	22	156	1916929	1	0
70	121728	1949382	16.01	105098	3982	15702	49424	60	0
72	130228	2032536	15.61	130228	1100	4495	350	28	0
72	62451	2034917	32.58	90	182	288	108	36	0
73	1048576	2097149	2.00	786432	7	9	2097149	3	1
74	525825	2100225	3.99	2298	50	57	3584	3	0
75	90449	2259087	24.98	1131	380	680	8632	29	0
76	143571	2424822	16.89	84195	108	538	1	147	0
77	715176	2766523	3.87	10	10	19	41	4	0
78	204316	2846228	13.93	204290	17	329	1967432	9	0
79	83334	3046907	36.56	2431	301	694	1574941	66	0
80	1000005	3105536	3.11	504865	4394	11141	222	370	0
81	403394	3387388	8.40	401861	11089	30204	3387389	10	955
82	916428	5105039	5.57	733811	143	345	5105040	188	176974
83	832081	5797879	6.97	828753	2416	8402	4361273	8	0
84	217918	5871175	26.94	9183	662	1214	5592868	78	0
85	853761	5947651	6.97	850178	349	1501	4558272	6	0
86	1270432	8814880	6.94	27	4	28	5	4	0
87	1048576	9424900	8.99	9	3	28	3	3	0
88	1088958	15392990	14.14	967799	41773	286900	8119845	37	0

Table A.3: Code and Data Size in MB. For `Stencil` and `CSRbyNZ`, we use split-by-pattern. For `GenOSKI`, we use the split-by-count approach. In all the cases, we generate the code for 4 threads.

ID	CSR		Stencil		GenOSKI4		GenOSKI5		Unfolding		CSRbyNZ	
	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data
1	0.002	0.088	0.062	0.031	0.053	0.044	0.106	0.042	0.098	0.028	0.000	0.047
2	0.001	0.063	0.001	0.036	0.005	0.045	0.004	0.039	0.018	0.036	0.001	0.053
3	0.001	0.147	0.008	0.084	0.005	0.096	0.021	0.099	0.096	0.041	0.003	0.123
4	0.001	0.187	0.296	0.100	0.170	0.151	0.256	0.137	0.276	0.063	0.139	0.159
5	0.001	0.276	0.352	0.106	0.104	0.197	0.154	0.193	0.355	0.109	0.050	0.181
6	0.001	0.195	0.025	0.116	0.018	0.149	0.028	0.141	0.081	0.007	0.004	0.170
7	0.001	0.245	0.360	0.110	0.038	0.217	0.053	0.216	0.361	0.113	0.106	0.180
8	0.001	0.213	0.333	0.124	0.217	0.141	0.381	0.138	0.342	0.113	0.040	0.191
9	0.001	0.302	0.438	0.152	0.239	0.193	0.292	0.193	0.435	0.114	0.003	0.246
10	0.001	0.288	0.140	0.170	0.219	0.194	0.354	0.185	0.437	0.135	0.035	0.250
11	0.001	0.309	0.019	0.183	0.018	0.237	0.076	0.223	0.220	0.003	0.003	0.269
12	0.001	0.443	0.597	0.178	0.121	0.332	0.232	0.326	0.608	0.181	0.089	0.307
13	0.001	0.290	0.463	0.187	0.316	0.232	0.414	0.220	0.552	0.190	0.165	0.281
14	0.001	0.313	0.635	0.204	0.123	0.223	0.179	0.218	0.617	0.181	0.068	0.308
15	0.001	0.391	0.420	0.250	0.333	0.284	0.554	0.276	0.679	0.136	0.062	0.378
16	0.001	0.623	0.895	0.305	0.129	0.559	0.233	0.554	0.900	0.308	0.035	0.476
17	0.001	0.522	0.954	0.327	0.770	0.380	0.967	0.367	0.930	0.245	0.121	0.499
18	0.001	0.583	0.666	0.369	0.818	0.419	1.086	0.406	1.046	0.258	0.074	0.556
19	0.001	0.664	0.604	0.424	0.419	0.489	0.905	0.469	1.234	0.350	0.063	0.636
20	0.001	0.757	1.305	0.464	0.784	0.537	1.438	0.526	1.245	0.171	0.042	0.711
21	0.001	1.152	1.434	0.499	0.038	0.887	0.113	0.878	1.457	0.502	0.059	0.777
22	0.001	0.827	0.047	0.531	0.031	0.582	1.331	0.567	1.447	0.433	0.031	0.790
23	0.001	0.859	0.468	0.554	0.377	0.629	0.897	0.607	1.591	0.486	0.071	0.829
24	0.001	0.904	1.243	0.579	0.182	0.647	0.950	0.634	1.656	0.488	0.058	0.875
25	0.001	1.103	1.687	0.625	1.358	0.798	1.869	0.755	1.769	0.459	0.030	0.967
26	0.001	0.989	0.799	0.632	0.151	0.679	0.474	0.666	1.723	0.602	0.051	0.949
27	0.001	1.050	0.194	0.661	0.152	0.767	0.074	0.716	1.795	0.635	0.026	0.982
28	0.001	1.103	1.407	0.708	0.752	0.838	1.613	0.797	2.095	0.526	0.090	1.058

Table A.3 continued: Code and Data Size in MB.

ID	CSR		Stencil		GenOSKI4		GenOSKI5		Unfolding		CSRbyNZ	
	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data
29	0.001	1.423	2.168	0.713	0.113	1.413	0.149	1.409	2.179	0.716	0.176	1.140
30	0.001	1.208	0.066	0.761	0.227	0.859	0.372	0.834	1.971	0.460	0.013	1.129
31	0.001	1.405	2.272	0.756	0.223	0.947	0.751	0.965	1.921	0.423	0.269	1.202
32	0.001	1.243	0.058	0.798	0.025	0.876	1.832	0.853	2.172	0.650	0.032	1.188
33	0.001	1.313	2.433	0.794	0.289	1.456	0.636	1.423	2.471	0.794	1.030	1.209
34	0.001	1.297	0.848	0.825	0.174	0.883	0.303	0.867	2.216	0.784	0.040	1.236
35	0.001	1.700	2.538	0.814	0.189	1.567	0.348	1.550	1.348	0.003	0.956	1.322
36	0.001	1.345	2.373	0.855	0.247	0.931	0.522	0.912	2.389	0.851	0.124	1.295
37	0.001	1.539	2.751	0.904	0.149	1.756	0.323	1.740	2.769	0.907	0.882	1.393
38	0.001	1.577	0.544	1.020	0.256	1.150	1.058	1.116	2.956	0.946	0.077	1.525
39	0.001	1.622	0.157	1.045	0.243	1.124	0.386	1.093	2.806	0.760	0.042	1.559
40	0.001	1.764	0.251	1.143	0.353	1.249	1.048	1.214	3.117	0.430	0.078	1.708
41	0.001	3.778	2.117	1.581	0.391	2.368	1.009	2.324	4.851	1.238	0.002	2.312
42	0.002	4.356	2.133	1.618	0.395	2.416	1.039	2.368	4.953	1.265	0.002	2.362
43	0.001	2.307	0.157	1.449	0.267	1.736	0.360	1.637	4.061	1.203	0.014	2.150
44	0.001	2.663	4.207	1.410	0.697	2.328	1.186	2.264	4.272	1.408	0.284	2.239
45	0.001	2.553	4.373	1.511	0.190	2.965	0.386	2.946	4.400	1.514	0.483	2.328
46	0.001	2.919	0.003	1.689	0.032	2.072	0.046	1.964	0.190	0.938	0.003	2.456
47	0.001	2.795	0.135	1.737	0.146	2.123	0.667	1.987	4.647	0.700	0.016	2.569
48	0.001	2.683	4.117	1.687	0.226	1.860	2.071	1.827	4.680	1.488	0.096	2.557
49	0.001	4.216	6.285	2.119	1.881	3.169	2.652	3.063	6.373	2.110	2.264	3.227
50	0.001	3.649	0.358	2.369	0.143	2.665	0.275	2.572	6.796	2.088	0.082	3.538
51	0.001	8.853	11.06	3.264	0.220	6.193	0.520	6.159	11.76	3.212	1.669	5.666
52	0.001	5.351	9.220	3.216	0.181	6.347	0.409	6.330	9.270	3.219	0.555	4.947
53	0.001	5.630	0.150	3.603	0.003	3.906	0.271	3.982	9.684	2.881	0.022	5.361
54	0.001	6.981	11.50	3.890	1.296	6.872	2.724	6.701	11.53	3.890	1.851	6.053
55	0.001	7.092	11.27	3.942	0.546	7.371	1.080	7.294	4.919	0.004	1.322	6.077
56	0.001	6.598	0.429	4.287	0.141	4.820	0.305	4.653	12.28	3.789	0.080	6.400
57	0.001	7.411	5.385	4.777	1.392	6.060	2.850	5.689	13.56	1.768	0.041	7.156
58	0.001	8.905	18.76	5.843	0.850	6.852	3.106	6.570	19.14	5.622	0.472	8.800
59	0.001	12.21	18.65	6.109	0.682	10.07	2.833	9.560	9.220	0.003	0.002	9.928

Table A.3 continued: Code and Data Size in MB.

ID	CSR		Stencil		GenOSKI4		GenOSKI5		Unfolding		CSRbyNZ	
	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data	Code	Data
60	0.001	13.59	0.612	8.315	0.047	9.631	0.090	9.301	2.362	8.004	0.004	12.26
61	0.001	12.74	13.30	8.354	1.938	9.258	4.086	9.004	24.79	5.568	0.294	12.55
62	0.001	17.72	0.241	10.50	0.241	15.87	0.445	15.07	26.18	6.952	0.021	15.36
63	0.001	17.43	28.29	10.40	1.570	13.15	6.048	13.30	28.57	9.820	0.031	15.96
64	0.001	22.10	33.07	11.44	2.220	14.44	4.234	13.95	30.75	11.50	2.546	17.65
65	0.001	20.24	21.11	12.49	3.260	14.75	6.237	14.21	21.76	11.50	0.034	18.83
66	0.001	22.49	36.28	12.90	1.909	21.20	3.772	20.79	37.50	12.81	0.032	20.00
67	0.001	22.99	0.316	14.74	0.147	15.80	0.429	15.38	38.15	4.524	0.028	21.95
68	0.001	22.68	1.969	14.80	0.019	16.56	0.269	16.41	40.42	7.019	0.014	22.12
69	0.001	22.68	1.969	14.80	0.019	16.56	0.269	16.41	45.38	14.62	0.014	22.12
70	0.001	24.16	42.66	14.92	5.085	18.77	17.05	17.48	42.47	11.22	0.135	22.77
71	0.001	25.24	44.38	15.50	0.815	21.33	3.871	21.38	25.43	0.297	0.051	23.75
72	0.001	24.24	0.189	15.76	0.368	17.35	0.712	16.85	33.71	0.134	0.075	23.52
73	0.001	40.00	54.17	16.00	0.002	20.00	0.003	20.00	55.74	16.00	0.002	28.00
74	0.001	32.05	0.384	18.02	0.015	20.97	0.021	19.99	6.045	12.37	0.002	26.04
75	0.001	27.23	0.768	17.57	0.634	18.85	2.499	18.37	44.94	5.728	0.045	26.19
76	0.001	29.94	49.72	18.68	0.152	21.79	1.388	21.16	11.21	0.006	0.800	28.29
77	0.001	42.57	0.004	23.83	0.006	27.57	0.032	27.37	7.870	3.507	0.002	34.38
78	0.001	35.69	61.84	21.71	0.024	26.71	0.634	28.91	61.48	21.27	0.013	33.35
79	0.001	36.14	3.701	23.55	0.284	25.93	0.829	25.21	69.10	22.91	0.175	35.18
80	0.001	50.79	71.98	23.88	2.960	30.76	5.771	29.60	28.89	0.930	5.071	39.35
81	0.001	44.92	73.69	25.84	8.363	41.70	14.90	40.87	73.83	25.84	0.009	40.30
82	0.001	72.40	114.3	38.94	0.082	77.85	0.164	77.84	114.3	38.95	0.677	61.24
83	0.001	79.04	126.6	44.24	2.227	62.97	6.866	60.69	125.6	40.63	0.008	69.52
84	0.001	70.51	10.41	45.57	0.560	48.96	1.573	47.63	126.1	44.21	0.223	68.02
85	0.001	81.09	130.6	45.39	0.170	62.79	1.183	59.96	130.7	40.90	0.005	71.32
86	0.001	120.2	0.010	72.09	0.003	83.96	0.019	88.36	23.92	0.161	0.005	105.7
87	0.001	123.8	0.008	75.90	0.002	89.84	0.024	83.09	5.944	0.091	0.004	111.8
88	0.001	192.7	378.8	117.9	58.80	165.0	151.1	157.1	342.3	104.0	0.075	180.3

Table A.4: Speedup for all methods for loome2 with respect to `MKL`. All the methods (including `MKL`) run with 4 threads.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
1	1.137	1.262	1.516	1.514	1.974	1.482	Unfolding	1.974
2	1.312	1.521	1.196	1.302	1.938	1.467	Unfolding	1.938
3	1.138	1.382	1.143	1.112	1.799	1.038	Unfolding	1.799
4	1.493	1.975	1.566	1.621	2.277	1.763	Unfolding	2.277
5	1.039	2.569	1.735	1.727	2.460	1.948	stencil	2.569
6	1.143	1.438	1.113	1.151	2.250	1.216	Unfolding	2.250
7	0.936	2.588	1.835	1.813	2.355	2.228	stencil	2.588
8	1.095	1.775	1.544	1.493	1.668	1.414	stencil	1.775
9	1.059	1.413	1.330	1.191	1.484	1.489	CSRbyNZ	1.489
10	1.116	1.481	1.154	1.037	1.063	1.078	stencil	1.481
11	1.072	1.387	1.068	1.057	1.954	1.326	Unfolding	1.954
12	1.056	1.634	1.502	1.236	1.660	1.752	CSRbyNZ	1.752
13	1.299	1.000	0.945	0.897	0.963	0.932	CSR	1.299
14	1.141	0.713	1.098	1.109	0.768	0.746	CSR	1.141
15	1.088	0.809	0.936	0.862	0.937	0.990	CSR	1.088
16	1.010	1.084	1.074	1.041	1.065	1.442	CSRbyNZ	1.442
17	1.074	0.823	0.920	0.817	0.975	1.070	CSR	1.074
18	1.044	0.714	0.629	0.569	0.583	0.933	CSR	1.044
19	1.087	0.796	0.917	0.665	0.625	0.921	CSR	1.087
20	0.974	0.731	0.831	0.640	0.857	1.106	CSRbyNZ	1.106
21	1.097	1.066	1.183	1.129	1.074	1.464	CSRbyNZ	1.464
22	1.051	1.358	1.142	0.588	0.630	0.968	stencil	1.358
23	1.088	0.945	0.963	0.744	0.608	0.908	CSR	1.088
24	1.045	0.602	0.934	0.679	0.571	0.937	CSR	1.045
25	0.952	1.024	1.006	0.918	1.129	1.275	CSRbyNZ	1.275
26	1.128	0.855	1.287	1.098	0.690	1.009	genOSKI4	1.287
27	1.060	1.227	1.061	1.291	0.677	1.048	genOSKI5	1.291
28	1.034	0.640	0.744	0.593	0.596	0.836	CSR	1.034
29	0.965	1.382	1.469	1.438	1.362	1.642	CSRbyNZ	1.642
30	1.032	1.227	0.964	0.987	0.756	1.057	stencil	1.227

Table A.4 continued: Speedups with respect to MKL in loome2.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
31	1.370	1.183	1.577	1.297	1.383	1.673	CSRbyNZ	1.673
32	1.051	1.356	1.184	0.620	0.650	0.978	stencil	1.356
33	1.100	0.992	1.078	1.063	1.009	0.985	CSR	1.100
34	1.064	0.886	1.206	1.115	0.654	0.973	genOSKI4	1.206
35	1.145	1.511	1.563	1.521	2.353	1.525	Unfolding	2.353
36	1.019	0.580	1.040	0.983	0.582	0.826	genOSKI4	1.040
37	1.110	1.103	1.166	1.159	1.090	1.291	CSRbyNZ	1.291
38	1.097	0.968	1.019	0.830	0.590	0.904	CSR	1.097
39	1.031	1.155	1.030	1.026	0.630	0.926	stencil	1.155
40	1.017	1.006	0.961	0.823	0.649	0.858	CSR	1.017
41	1.040	0.689	0.638	0.637	0.826	1.628	CSRbyNZ	1.628
42	1.083	0.710	0.645	0.644	0.869	1.642	CSRbyNZ	1.642
43	1.048	1.210	0.828	0.982	0.605	1.046	stencil	1.210
44	1.170	1.425	1.529	1.466	1.309	1.873	CSRbyNZ	1.873
45	1.098	0.944	1.180	1.172	1.054	1.415	CSRbyNZ	1.415
46	1.100	1.514	1.037	1.071	1.893	1.209	Unfolding	1.893
47	0.996	1.172	0.814	0.874	0.671	0.990	stencil	1.172
48	1.067	0.652	1.091	0.798	0.587	0.951	genOSKI4	1.091
49	1.159	0.440	0.917	0.856	0.429	0.825	CSR	1.159
50	1.001	1.077	0.970	0.977	0.217	0.861	stencil	1.077
51	1.482	0.711	1.138	1.110	0.703	1.423	CSR	1.482
52	1.111	0.368	0.896	0.869	0.365	1.212	CSRbyNZ	1.212
53	1.052	1.255	1.331	1.038	0.202	1.039	genOSKI4	1.331
54	1.254	0.658	1.185	1.023	0.649	1.428	CSRbyNZ	1.428
55	1.245	0.627	1.121	1.075	2.925	1.563	Unfolding	2.925
56	1.082	1.346	1.214	1.207	0.185	0.996	stencil	1.346
57	1.071	0.460	0.813	0.617	0.251	1.075	CSRbyNZ	1.075
58	1.042	0.231	1.280	0.762	0.229	0.880	genOSKI4	1.280
59	1.125	0.492	0.562	0.558	1.195	1.137	Unfolding	1.195
60	1.049	1.762	1.303	1.372	1.368	1.059	stencil	1.762
61	1.056	0.537	1.162	0.929	0.388	1.027	genOSKI4	1.162
62	1.052	1.307	0.643	0.671	0.561	1.054	stencil	1.307

Table A.4 continued: Speedups with respect to MKL in loome2.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
63	1.231	0.548	1.073	0.861	0.546	1.265	CSRbyNZ	1.265
64	1.056	0.609	1.122	1.038	0.637	1.014	genOSKI4	1.122
65	1.042	0.602	1.021	0.903	0.612	1.015	CSR	1.042
66	1.248	0.703	0.944	0.917	0.695	1.418	CSRbyNZ	1.418
67	1.009	1.630	1.321	1.349	0.550	1.092	stencil	1.630
68	1.008	1.388	1.352	1.289	0.478	1.023	stencil	1.388
69	1.015	1.390	1.353	1.315	0.382	1.025	stencil	1.390
70	1.071	0.448	0.806	0.639	0.482	1.044	CSR	1.071
71	1.078	0.462	0.977	0.753	0.993	1.072	CSR	1.078
72	1.019	1.634	1.141	1.209	0.718	1.043	stencil	1.634
73	1.085	0.703	0.809	0.715	0.690	1.149	CSRbyNZ	1.149
74	1.065	1.268	0.675	0.704	1.285	1.047	Unfolding	1.285
75	1.025	1.515	1.154	1.190	0.543	1.053	stencil	1.515
76	1.361	0.611	1.460	1.570	3.236	1.331	Unfolding	3.236
77	1.060	1.235	0.684	0.591	1.739	1.037	Unfolding	1.739
78	1.026	0.436	1.110	0.826	0.439	1.011	genOSKI4	1.110
79	1.014	1.249	1.254	1.270	0.400	0.994	genOSKI5	1.270
80	1.157	0.615	0.940	0.922	1.332	0.980	Unfolding	1.332
81	0.981	0.516	0.562	0.520	0.516	1.024	CSRbyNZ	1.024
82	1.030	0.628	0.467	0.411	0.629	0.926	CSR	1.030
83	1.042	0.590	0.382	0.417	0.599	1.081	CSRbyNZ	1.081
84	1.033	1.183	1.087	1.112	0.436	0.995	stencil	1.183
85	1.035	0.605	0.340	0.408	0.612	1.079	CSRbyNZ	1.079
86	1.045	1.262	0.797	0.480	2.106	1.022	Unfolding	2.106
87	1.022	1.268	0.880	0.599	2.902	1.028	Unfolding	2.902
88	1.033	0.304	0.205	0.203	0.336	0.744	CSR	1.033
Avg	1.090	1.036	1.055	0.974	1.008	1.162		1.453

Table A.5: Speedup for all methods for loome3 with respect to `MKL`. All the methods (including `MKL`) run with 4 threads.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
1	1.104	0.921	1.150	1.119	1.353	1.223	Unfolding	1.353
2	1.209	1.401	1.025	1.215	1.733	1.512	Unfolding	1.733
3	1.189	1.600	1.336	1.311	1.616	1.373	Unfolding	1.616
4	1.179	1.158	1.056	1.022	1.345	1.385	CSRbyNZ	1.385
5	1.416	1.837	1.503	1.445	1.809	1.712	Stencil	1.837
6	1.108	1.498	1.162	1.188	1.972	1.401	Unfolding	1.972
7	1.258	1.705	1.495	1.465	1.618	1.781	CSRbyNZ	1.781
8	1.223	0.981	0.995	0.793	1.026	1.035	CSR	1.223
9	1.097	1.082	1.068	0.969	1.131	1.480	CSRbyNZ	1.480
10	1.184	1.240	0.954	0.857	0.939	1.138	Stencil	1.240
11	1.028	1.467	1.149	1.134	1.543	1.544	CSRbyNZ	1.544
12	1.124	1.611	1.411	1.294	1.602	1.683	CSRbyNZ	1.683
13	1.351	0.680	0.720	0.680	0.646	0.733	CSR	1.351
14	1.185	0.525	1.060	0.871	0.551	0.815	CSR	1.185
15	1.265	0.571	0.813	0.708	0.687	0.999	CSR	1.265
16	1.403	1.103	0.979	0.976	1.060	1.396	CSR	1.403
17	1.257	0.658	0.695	0.622	0.689	0.856	CSR	1.257
18	1.109	0.551	0.528	0.478	0.549	0.938	CSR	1.109
19	1.131	0.623	0.809	0.609	0.554	0.983	CSR	1.131
20	1.150	0.637	0.727	0.544	0.712	1.098	CSR	1.150
21	1.254	1.197	1.134	1.107	1.149	1.665	CSRbyNZ	1.665
22	1.098	1.189	1.200	0.557	0.560	1.076	genOSKI4	1.200
23	1.191	0.811	0.884	0.670	0.552	0.940	CSR	1.191
24	1.141	0.507	0.817	0.619	0.464	0.984	CSR	1.141
25	0.966	0.963	0.867	0.767	0.999	1.319	CSRbyNZ	1.319
26	1.202	0.774	1.324	1.040	0.613	1.072	genOSKI4	1.324
27	1.116	1.199	1.048	1.355	0.667	1.129	genOSKI5	1.355
28	1.091	0.535	0.653	0.530	0.524	0.884	CSR	1.091
29	0.816	1.277	1.453	1.360	1.224	1.684	CSRbyNZ	1.684
30	1.100	1.163	0.825	0.907	0.638	1.119	Stencil	1.163

Table A.5 continued: Speedups with respect to MKL in loome3.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
31	1.288	1.156	1.497	1.247	1.337	1.926	CSRbyNZ	1.926
32	1.114	1.129	1.214	0.494	0.557	1.064	genOSKI4	1.214
33	1.200	0.786	0.887	0.874	0.831	0.806	CSR	1.200
34	1.099	0.848	1.222	1.136	0.617	1.082	genOSKI4	1.222
35	1.085	1.522	1.659	1.559	2.500	1.715	Unfolding	2.500
36	1.149	0.556	1.082	1.012	0.554	0.883	CSR	1.149
37	1.057	0.915	1.017	0.995	0.911	1.128	CSRbyNZ	1.128
38	1.104	0.837	0.995	0.757	0.487	0.879	CSR	1.104
39	1.156	1.172	1.056	1.072	0.599	1.066	Stencil	1.172
40	1.151	0.944	0.965	0.794	0.496	0.957	CSR	1.151
41	1.156	0.816	1.220	1.130	0.669	2.120	CSRbyNZ	2.120
42	1.145	0.853	1.215	1.094	0.649	2.141	CSRbyNZ	2.141
43	1.048	1.144	0.791	0.949	0.437	1.168	CSRbyNZ	1.168
44	1.071	0.908	1.462	1.430	0.854	2.159	CSRbyNZ	2.159
45	1.006	0.703	1.170	1.143	0.684	1.453	CSRbyNZ	1.453
46	1.152	1.983	1.318	1.371	2.306	1.735	Unfolding	2.306
47	1.122	1.275	0.847	0.915	0.529	1.188	Stencil	1.275
48	1.190	0.374	1.164	0.654	0.311	1.086	CSR	1.190
49	1.281	0.384	0.873	0.661	0.370	0.702	CSR	1.281
50	1.076	0.975	0.992	0.977	0.143	0.890	CSR	1.076
51	1.556	0.859	1.574	1.479	0.848	1.431	genOSKI4	1.574
52	1.086	0.366	0.807	0.818	0.353	1.181	CSRbyNZ	1.181
53	0.940	1.241	1.558	1.092	0.202	1.031	genOSKI4	1.558
54	1.151	0.714	1.133	0.974	0.697	1.400	CSRbyNZ	1.400
55	1.179	0.696	1.199	1.097	2.794	1.556	Unfolding	2.794
56	1.069	1.889	1.742	1.699	0.278	1.083	Stencil	1.889
57	0.975	0.584	0.899	0.728	0.368	1.110	CSRbyNZ	1.110
58	0.967	0.314	1.220	0.844	0.312	0.910	genOSKI4	1.220
59	1.092	0.539	0.932	0.845	1.218	1.112	Unfolding	1.218
60	1.034	1.424	1.418	1.472	1.204	1.050	genOSKI5	1.472
61	0.996	0.574	1.126	0.953	0.414	1.005	genOSKI4	1.126
62	1.048	1.200	0.855	0.864	0.566	1.052	Stencil	1.200

Table A.5 continued: Speedups with respect to MKL in loome3.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
63	1.196	0.557	1.271	0.989	0.557	1.269	genOSKI4	1.271
64	1.033	0.601	1.002	0.953	0.627	1.011	CSR	1.033
65	1.028	0.599	1.120	0.880	0.614	1.015	genOSKI4	1.120
66	1.256	0.761	1.024	1.038	0.738	1.599	CSRbyNZ	1.599
67	1.018	1.512	1.431	1.484	0.542	1.014	Stencil	1.512
68	1.013	1.345	1.409	1.391	0.478	1.020	genOSKI4	1.409
69	1.011	1.343	1.412	1.390	0.385	1.021	genOSKI4	1.412
70	1.046	0.437	0.968	0.613	0.470	1.070	CSRbyNZ	1.070
71	1.051	0.456	1.071	0.916	0.971	1.077	CSRbyNZ	1.077
72	1.021	1.503	1.352	1.400	0.704	1.007	Stencil	1.503
73	1.054	0.653	0.705	0.615	0.616	1.090	CSRbyNZ	1.090
74	1.045	1.182	0.598	0.612	1.237	1.031	Unfolding	1.237
75	1.022	1.445	1.436	1.346	0.539	1.012	Stencil	1.445
76	1.255	0.583	1.676	1.575	2.855	1.312	Unfolding	2.855
77	1.050	1.134	0.647	0.560	1.615	1.019	Unfolding	1.615
78	1.000	0.448	1.050	0.718	0.453	1.023	genOSKI4	1.050
79	1.011	1.219	1.330	1.398	0.399	0.987	genOSKI5	1.398
80	1.224	0.620	0.854	0.852	1.394	1.036	Unfolding	1.394
81	0.980	0.546	0.469	0.437	0.547	1.066	CSRbyNZ	1.066
82	1.057	0.573	0.505	0.428	0.573	0.971	CSR	1.057
83	1.103	0.621	0.307	0.343	0.631	1.147	CSRbyNZ	1.147
84	1.011	1.148	0.978	1.011	0.427	0.973	Stencil	1.148
85	1.063	0.611	0.225	0.306	0.621	1.156	CSRbyNZ	1.156
86	1.036	1.188	0.763	0.468	2.052	1.028	Unfolding	2.052
87	1.032	1.252	0.857	0.568	3.195	1.037	Unfolding	3.195
88	1.034	0.259	0.172	0.174	0.286	0.648	CSR	1.034
Avg	1.117	0.952	1.052	0.952	0.899	1.189		1.437

Table A.6: Speedup for all methods for i2pc3 with respect to `MKL`. All the methods (including `MKL`) run with 4 threads.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
1	1.227	0.926	1.307	1.399	1.723	1.410	Unfolding	1.723
2	1.353	1.623	1.108	1.294	1.705	1.463	Unfolding	1.705
3	1.277	1.080	1.069	1.020	1.748	0.957	Unfolding	1.748
4	1.501	1.714	1.143	1.149	2.181	1.411	Unfolding	2.181
5	1.148	1.993	1.358	1.384	2.197	1.624	Unfolding	2.197
6	1.307	1.081	1.045	0.971	1.883	1.073	Unfolding	1.883
7	1.051	2.279	1.350	1.344	2.458	1.585	Unfolding	2.458
8	1.213	1.587	1.153	1.034	1.788	1.119	Unfolding	1.788
9	1.079	1.307	1.024	1.011	1.587	1.058	Unfolding	1.587
10	1.110	1.235	0.880	0.870	1.446	0.991	Unfolding	1.446
11	1.184	1.030	0.946	1.070	1.546	1.038	Unfolding	1.546
12	1.147	1.744	1.221	1.230	1.718	1.292	stencil	1.744
13	1.422	1.126	0.797	0.944	1.172	0.781	CSR	1.422
14	1.211	1.073	0.949	0.965	1.078	0.916	CSR	1.211
15	1.208	0.907	0.833	0.888	1.044	0.956	CSR	1.208
16	1.048	1.394	1.014	0.993	1.322	1.399	CSRbyNZ	1.399
17	1.156	1.086	0.897	0.896	1.090	0.891	CSR	1.156
18	1.154	0.794	0.616	0.679	0.813	0.712	CSR	1.154
19	1.163	0.865	0.794	0.810	0.869	0.799	CSR	1.163
20	1.075	0.995	0.779	0.783	1.069	0.981	CSR	1.075
21	1.112	1.334	1.146	1.065	1.257	1.153	stencil	1.334
22	1.087	1.207	1.063	0.695	0.835	0.853	stencil	1.207
23	1.198	1.061	0.838	0.787	0.790	0.779	CSR	1.198
24	1.133	0.818	0.942	0.793	0.752	0.830	CSR	1.133
25	0.989	1.267	1.049	1.056	1.508	0.941	Unfolding	1.508
26	1.162	1.040	1.184	1.044	0.901	0.854	genOSKI4	1.184
27	1.148	1.087	0.869	1.206	0.899	0.934	genOSKI5	1.206
28	1.108	0.777	0.754	0.701	0.770	0.743	CSR	1.108
29	0.994	1.506	1.424	1.341	1.578	1.215	Unfolding	1.578
30	1.053	1.081	0.782	0.764	0.937	0.910	stencil	1.081

Table A.6 continued: Speedups with respect to MKL in i2pc3.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
31	1.326	1.458	1.485	1.205	1.493	1.653	CSRbyNZ	1.653
32	1.099	1.104	1.066	0.743	0.790	0.889	stencil	1.104
33	1.134	1.276	0.950	0.968	1.295	1.107	Unfolding	1.295
34	1.085	0.933	1.143	1.030	0.773	0.845	genOSKI4	1.143
35	1.078	1.574	1.416	1.378	2.253	1.319	Unfolding	2.253
36	1.054	0.781	1.041	1.025	0.715	0.769	CSR	1.054
37	1.116	1.416	1.057	1.062	1.444	1.091	Unfolding	1.444
38	1.074	0.944	0.850	0.847	0.757	0.856	CSR	1.074
39	1.059	0.972	0.967	0.963	0.847	0.778	CSR	1.059
40	1.066	0.922	0.909	0.850	0.798	0.794	CSR	1.066
41	1.002	0.802	1.037	1.005	1.570	1.598	CSRbyNZ	1.598
42	1.052	0.852	1.057	1.037	1.501	1.647	CSRbyNZ	1.647
43	1.048	1.138	0.725	0.983	0.872	0.929	stencil	1.138
44	1.151	1.853	1.553	1.268	1.830	1.324	stencil	1.853
45	1.115	1.433	1.183	1.168	1.450	1.184	Unfolding	1.450
46	1.126	1.415	1.091	1.116	1.750	1.197	Unfolding	1.750
47	1.006	1.110	0.779	0.916	0.948	0.823	stencil	1.110
48	1.106	0.878	1.091	0.956	0.890	0.879	CSR	1.106
49	1.183	1.152	1.059	1.030	1.155	0.960	CSR	1.183
50	1.043	0.995	0.867	0.823	0.718	0.816	CSR	1.043
51	1.081	1.264	1.271	1.236	1.576	1.607	CSRbyNZ	1.607
52	1.037	1.089	0.983	0.971	1.104	0.954	Unfolding	1.104
53	0.989	1.053	1.188	0.926	0.816	0.968	genOSKI4	1.188
54	1.123	1.930	1.422	1.329	1.956	1.661	Unfolding	1.956
55	1.199	1.803	1.513	1.324	2.800	1.594	Unfolding	2.800
56	1.020	1.011	0.887	0.855	0.693	0.862	CSR	1.020
57	1.026	0.837	0.758	0.642	0.776	0.879	CSR	1.026
58	1.080	0.637	0.880	0.844	0.604	0.823	CSR	1.080
59	1.117	0.930	0.661	0.642	1.398	1.221	Unfolding	1.398
60	0.999	1.317	1.080	1.119	1.578	1.186	Unfolding	1.578
61	1.043	0.751	0.896	0.872	0.703	0.832	CSR	1.043
62	0.921	0.861	0.773	0.784	1.263	1.099	Unfolding	1.263

Table A.6 continued: Speedups with respect to MKL in i2pc3.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
63	1.184	1.151	1.095	1.020	0.857	1.024	CSR	1.184
64	0.977	1.207	1.219	1.114	1.279	1.094	Unfolding	1.279
65	0.972	0.926	0.825	0.745	1.088	0.954	Unfolding	1.088
66	1.267	1.256	1.100	1.130	1.238	1.219	CSR	1.267
67	1.000	1.200	0.990	1.011	0.874	0.930	stencil	1.200
68	1.009	0.938	0.949	0.794	0.774	0.957	CSR	1.009
69	1.009	0.955	0.962	0.786	0.540	0.969	CSR	1.009
70	0.971	0.851	0.753	0.684	0.952	0.874	CSR	0.971
71	1.143	0.707	0.805	0.692	1.558	1.099	Unfolding	1.558
72	1.000	1.143	0.807	0.823	0.985	0.894	stencil	1.143
73	1.098	0.559	1.491	1.409	0.472	1.460	genOSKI4	1.491
74	1.065	1.486	1.100	1.093	1.611	1.256	Unfolding	1.611
75	0.996	0.968	0.925	0.921	0.854	0.887	CSR	0.996
76	1.584	1.014	1.378	1.365	3.886	1.310	Unfolding	3.886
77	1.099	1.493	1.194	1.180	1.886	1.353	Unfolding	1.886
78	0.840	0.298	1.005	0.841	0.278	0.915	genOSKI4	1.005
79	1.016	1.056	0.903	0.909	0.180	0.874	stencil	1.056
80	1.104	0.321	1.208	0.983	1.738	1.164	Unfolding	1.738
81	1.415	0.379	0.927	0.891	0.323	1.761	CSRbyNZ	1.761
82	0.910	0.446	0.787	0.857	0.462	1.043	CSRbyNZ	1.043
83	0.862	0.284	0.834	0.727	0.283	0.934	CSRbyNZ	0.934
84	0.722	1.130	0.983	1.049	0.128	0.876	stencil	1.130
85	0.907	0.336	0.783	0.772	0.338	1.014	CSRbyNZ	1.014
86	1.043	1.577	1.307	1.164	4.126	1.070	Unfolding	4.126
87	0.986	1.651	1.291	1.466	4.517	1.007	Unfolding	4.517
88	1.057	0.138	0.620	0.410	0.157	1.040	CSR	1.057
Avg	1.100	1.102	1.025	0.988	1.263	1.077		1.470

Table A.7: Speedup for all methods for i2pc5 with respect to `MKL`. All the methods (including `MKL`) run with 4 threads.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
1	1.146	0.796	1.193	1.142	1.689	1.267	Unfolding	1.689
2	1.285	1.373	1.128	1.088	1.410	1.131	Unfolding	1.410
3	1.227	0.860	1.016	0.996	1.584	0.849	Unfolding	1.584
4	1.420	1.713	1.021	1.022	1.831	1.190	Unfolding	1.831
5	1.052	1.923	1.174	1.129	1.927	1.412	Unfolding	1.927
6	1.216	0.900	0.946	0.897	1.667	0.941	Unfolding	1.667
7	0.968	2.150	1.206	1.067	2.114	1.436	stencil	2.150
8	1.137	1.343	0.921	0.746	1.537	0.943	Unfolding	1.537
9	1.026	1.202	1.026	0.851	1.494	1.003	Unfolding	1.494
10	1.069	1.153	0.811	0.784	1.258	0.860	Unfolding	1.258
11	1.159	1.152	0.820	1.002	1.651	0.911	Unfolding	1.651
12	1.073	1.427	1.100	1.078	1.545	1.104	Unfolding	1.545
13	1.352	1.100	0.643	0.728	1.122	0.667	CSR	1.352
14	1.115	1.086	0.956	0.827	1.086	0.908	CSR	1.115
15	1.229	0.826	0.776	0.804	1.140	0.879	CSR	1.229
16	1.041	1.233	0.922	0.897	1.223	0.944	stencil	1.233
17	1.061	0.877	0.763	0.732	0.944	0.779	CSR	1.061
18	1.134	0.717	0.566	0.643	0.745	0.646	CSR	1.134
19	1.076	0.832	0.714	0.685	0.746	0.717	CSR	1.076
20	1.043	0.888	0.675	0.678	0.889	0.894	CSR	1.043
21	1.012	1.110	1.016	0.980	1.141	1.004	Unfolding	1.141
22	1.055	1.081	0.943	0.597	0.735	0.734	stencil	1.081
23	1.141	0.908	0.749	0.719	0.689	0.725	CSR	1.141
24	1.115	0.764	0.894	0.655	0.718	0.668	CSR	1.115
25	0.973	1.150	0.943	0.956	1.269	0.808	Unfolding	1.269
26	1.118	0.916	1.008	0.920	0.764	0.732	CSR	1.118
27	1.071	0.957	0.774	1.098	0.846	0.867	genOSKI5	1.098
28	1.044	0.713	0.684	0.639	0.699	0.667	CSR	1.044
29	0.998	1.429	1.315	1.257	1.455	1.099	Unfolding	1.455
30	1.034	0.918	0.702	0.805	0.840	0.772	CSR	1.034

Table A.7 continued: Speedups with respect to MKL in i2pc5.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
31	1.309	1.332	1.363	1.130	1.435	1.518	CSRbyNZ	1.518
32	1.052	0.957	0.959	0.649	0.689	0.836	CSR	1.052
33	1.098	1.205	0.890	0.930	1.271	0.967	Unfolding	1.271
34	1.105	0.866	1.058	0.878	0.799	0.785	CSR	1.105
35	1.071	1.479	1.344	1.283	2.050	1.240	Unfolding	2.050
36	1.043	0.717	0.981	0.936	0.644	0.716	CSR	1.043
37	1.114	1.342	1.038	0.935	1.352	1.063	Unfolding	1.352
38	1.111	0.919	0.776	0.780	0.712	0.768	CSR	1.111
39	1.059	0.860	0.896	0.859	0.780	0.705	CSR	1.059
40	1.040	0.790	0.831	0.757	0.748	0.742	CSR	1.040
41	1.007	0.728	0.973	0.960	1.362	1.357	Unfolding	1.362
42	1.008	0.763	0.983	0.942	1.361	1.439	CSRbyNZ	1.439
43	1.008	1.008	0.674	0.942	0.762	0.860	CSR	1.008
44	1.129	1.576	1.257	1.173	1.833	1.143	Unfolding	1.833
45	1.112	1.281	1.107	1.062	1.335	1.076	Unfolding	1.335
46	1.082	1.384	1.071	1.088	1.641	1.125	Unfolding	1.641
47	1.016	0.944	0.770	0.900	0.873	0.779	CSR	1.016
48	1.086	0.839	1.058	0.887	0.852	0.807	CSR	1.086
49	1.207	1.144	1.066	1.031	1.144	0.925	CSR	1.207
50	1.034	0.958	0.796	0.733	0.674	0.768	CSR	1.034
51	1.060	1.224	1.258	1.196	1.547	1.462	Unfolding	1.547
52	1.045	1.046	0.975	0.935	1.052	0.944	Unfolding	1.052
53	1.015	1.045	1.179	0.880	0.769	0.944	genOSKI4	1.179
54	1.118	1.844	1.349	1.327	1.909	1.559	Unfolding	1.909
55	1.158	1.650	1.406	1.275	2.583	1.458	Unfolding	2.583
56	1.026	0.979	0.880	0.802	0.664	0.821	CSR	1.026
57	1.042	0.780	0.732	0.613	0.741	0.847	CSR	1.042
58	1.077	0.608	0.839	0.760	0.576	0.794	CSR	1.077
59	1.042	0.915	0.667	0.628	1.365	1.211	Unfolding	1.365
60	0.958	1.261	1.055	1.094	1.485	1.087	Unfolding	1.485
61	1.037	0.729	0.889	0.843	0.676	0.801	CSR	1.037
62	0.922	0.779	0.735	0.747	1.143	1.034	Unfolding	1.143

Table A.7 continued: Speedups with respect to MKL in i2pc5.

ID	CSR	stencil	genOSKI4	genOSKI5	Unfolding	CSRbyNZ	BestMTD	BestSpeed
63	1.183	1.115	1.116	0.993	0.956	1.010	CSR	1.183
64	0.964	1.092	1.154	1.093	0.922	1.083	genOSKI4	1.154
65	1.006	0.889	0.801	0.739	0.902	0.884	CSR	1.006
66	1.309	0.855	1.115	1.067	0.878	1.223	CSR	1.309
67	1.003	1.151	0.987	0.955	0.851	0.919	stencil	1.151
68	1.010	0.897	0.943	0.772	0.755	0.960	CSR	1.010
69	1.004	0.888	0.945	0.770	0.509	0.947	CSR	1.004
70	0.963	0.682	0.705	0.613	0.599	0.832	CSR	0.963
71	1.160	0.608	0.778	0.656	1.455	1.013	Unfolding	1.455
72	1.016	1.143	0.791	0.812	0.945	0.895	stencil	1.143
73	1.203	0.606	1.673	1.601	0.507	1.608	genOSKI4	1.673
74	1.082	1.455	1.127	1.142	1.662	1.249	Unfolding	1.662
75	1.012	0.874	0.912	0.880	0.504	0.865	CSR	1.012
76	1.668	0.625	1.406	1.392	3.884	1.295	Unfolding	3.884
77	1.009	1.461	1.175	1.117	1.819	1.294	Unfolding	1.819
78	0.853	0.270	1.026	0.831	0.228	0.891	genOSKI4	1.026
79	1.086	1.148	0.980	0.967	0.173	0.858	stencil	1.148
80	1.252	0.337	1.308	1.261	1.659	1.234	Unfolding	1.659
81	1.005	0.285	0.799	0.794	0.276	1.329	CSRbyNZ	1.329
82	0.967	0.443	0.862	0.830	0.440	1.102	CSRbyNZ	1.102
83	1.066	0.344	0.964	0.884	0.342	1.109	CSRbyNZ	1.109
84	1.051	1.688	1.102	1.368	0.181	1.132	stencil	1.688
85	0.996	0.352	0.759	0.762	0.351	1.119	CSRbyNZ	1.119
86	1.017	1.402	1.268	1.216	4.418	1.023	Unfolding	4.418
87	1.047	1.610	1.289	1.504	5.058	1.047	Unfolding	5.058
88	1.047	0.099	0.547	0.390	0.107	1.021	CSR	1.047
Avg	1.086	1.020	0.975	0.929	1.181	1.001		1.416

Table A.8: Speedup of split-by-pattern vs split-by-count for CSR and stencil. Split-by-pattern is faster for values larger than 1. Split-by-count if faster for values smaller than 1.

ID	loome2		loome3		i2pc3		i2pc5	
	CSRbyNZ	Stencil	CSRbyNZ	Stencil	CSRbyNZ	Stencil	CSRbyNZ	Stencil
1	0.98	0.7	0.93	0.62	0.96	0.42	0.93	0.4
2	1.37	1	1.2	1.05	1.28	1.02	1.16	0.86
3	0.99	0.89	1.01	0.88	1.04	0.94	1.06	0.98
4	0.96	1.01	1.13	1	0.78	1.04	0.75	1.09
5	1.01	1.01	0.98	1.02	0.73	0.82	0.7	0.71
6	0.99	1.01	1	1.03	0.95	0.99	0.9	0.93
7	0.69	1.05	0.65	0.93	0.62	0.88	0.47	0.67
8	0.72	1.1	0.87	1.01	0.57	0.85	0.53	0.72
9	0.99	0.93	0.89	0.97	0.99	0.92	0.93	0.89
10	0.93	0.83	0.95	0.97	0.66	0.72	0.66	0.6
11	1.18	0.97	1.09	0.94	1.1	0.93	1.13	0.7
12	0.99	1.16	0.97	1.02	0.85	0.85	0.8	0.95
13	0.91	0.95	1.06	0.97	0.71	0.77	0.7	0.64
14	1.26	0.98	0.81	0.93	0.52	0.97	0.41	0.95
15	0.97	0.98	1.06	0.95	0.82	0.79	0.66	0.72
16	1.1	0.97	0.98	1.01	0.7	0.95	0.83	1
17	1.37	1.04	1.44	1.03	0.72	0.95	0.62	0.98
18	0.98	1.2	1.04	1.03	0.98	0.91	0.97	0.89
19	1.04	0.98	1.14	1	0.89	0.95	0.91	0.82
20	0.89	0.99	0.96	1.01	0.68	0.95	0.67	0.98
21	1.05	1	0.97	1.02	0.95	1.11	0.92	0.99
22	0.97	0.95	0.94	1.16	0.91	0.77	0.9	0.74
23	0.99	1.1	1.02	1.21	1.04	0.87	1.07	0.88
24	1.15	0.97	1.18	0.98	0.95	0.9	1	0.93
25	1.32	1	1.03	1.01	0.91	0.98	0.87	0.84
26	1.01	1.16	1	1.21	0.91	1.09	0.92	1.07
27	1.14	1.08	1.12	1.14	1.07	0.94	0.92	0.94
28	1.07	1.1	1.09	1.07	1.02	0.96	1.04	0.93

Table A.8 continued: Speedup of split-by-pattern vs split-by-count for CSR and stencil.

ID	loome2		loome3		i2pc3		i2pc5	
	CSRbyNZ	Stencil	CSRbyNZ	Stencil	CSRbyNZ	Stencil	CSRbyNZ	Stencil
29	1.04	0.97	1.02	1.06	0.96	0.94	0.94	0.91
30	1.09	1.08	1.03	1.12	0.95	0.75	1	0.79
31	0.98	1.01	0.97	1.01	1	1.03	0.99	1
32	1.01	0.92	0.99	2.87	0.94	0.95	0.96	0.96
33	1.1	0.98	1.02	0.99	0.74	0.95	0.8	0.93
34	1.08	1.04	1.11	1.09	0.96	1	0.97	0.99
35	0.99	0.95	0.96	1.59	1	0.96	0.95	0.96
36	1.07	0.95	1.07	0.65	0.92	1.03	0.88	1
37	1.12	0.99	1.13	1.07	0.84	0.99	0.77	0.96
38	1.06	1.17	0.99	1.21	0.9	1	1.06	0.92
39	1.09	1.12	1.06	1.14	1.02	0.96	1	1.01
40	1.04	1.37	1.06	1.45	0.96	1.1	0.93	0.95
41	0.83	0.81	0.99	0.98	0.87	0.75	0.93	0.74
42	0.85	0.82	0.93	0.92	0.87	0.7	0.88	0.63
43	0.96	0.99	0.9	0.96	1.01	0.95	1.01	0.95
44	1	0.85	1.01	1.01	1.02	0.98	0.97	1.03
45	1.15	0.95	1.23	1.42	1.22	0.95	1.19	0.99
46	1.01	1.02	0.99	0.98	1.03	1.05	1.01	1
47	0.99	0.94	0.97	0.98	1.03	0.72	0.95	0.82
48	1.09	1.14	1.12	0.98	0.84	0.97	1.08	0.97
49	1.58	1.05	1.75	0.95	0.97	0.94	0.97	0.96
50	1	1.07	0.97	1.11	1.03	1.04	1.01	1.11
51	1.1	0.99	1.08	0.95	0.92	0.97	0.92	0.94
52	1.15	0.99	1.2	1.02	1	1.02	0.96	0.98
53	1	0.97	0.93	0.93	1	1	1.02	0.96
54	1.14	1.01	1.18	1.01	1.05	1.05	1.02	1.04
55	1.29	0.99	1.3	0.99	0.95	0.95	1	1.01
56	1.01	1.04	1.04	1.19	0.94	1.03	1.02	0.98
57	0.96	1.03	1.14	1	0.97	0.97	0.99	0.99
58	1.17	1	1.09	1	1.04	1	1.06	1.02
59	1.01	1	1.01	1	1	1.01	1	1.01

Table A.8 continued: Speedup of split-by-pattern vs split-by-count for CSR and stencil.

ID	loome2		loome3		i2pc3		i2pc5	
	CSRbyNZ	Stencil	CSRbyNZ	Stencil	CSRbyNZ	Stencil	CSRbyNZ	Stencil
60	1	1.05	1.02	1	0.94	1.01	1	1.02
61	1.07	1.04	1.06	1.04	1.03	0.99	1.09	0.98
62	1.02	0.84	0.89	1.03	0.77	0.99	0.74	1.02
63	1.02	1	1	1	0.89	1.01	0.85	0.86
64	1.01	1	0.97	1	1.01	1.02	1.05	1.28
65	0.99	1	0.96	1.01	1.04	0.99	1.01	0.87
66	0.98	0.99	0.88	0.98	0.96	0.98	0.95	0.98
67	1.07	1.06	0.97	1.06	0.99	0.98	0.98	1
68	1	1.39	1	1.38	1	1.03	0.99	1.02
69	1	1.39	1	1.37	1.02	1.03	1	0.99
70	1.03	1	1.07	1	0.95	1.03	0.94	1.31
71	1.01	1	0.99	1	0.93	1.12	0.97	0.99
72	1.03	1.07	0.98	1.05	1	0.95	1.01	0.89
73	1.05	1	1.01	1	0.9	1.11	1.11	1.1
74	1.01	1.01	1	1.01	0.97	0.93	1.01	1
75	1.04	1.09	0.97	1.08	0.95	0.94	0.93	0.97
76	1.02	1.02	1.04	1.02	0.95	0.8	0.94	0.82
77	0.98	0.99	0.99	0.99	0.83	0.95	0.87	0.99
78	0.98	1	0.96	1	0.95	1	0.96	0.92
79	0.95	1.09	0.94	1.08	0.95	1.02	1	1.11
80	0.98	0.94	0.95	0.94	0.97	0.86	1.16	0.94
81	0.97	1	0.96	0.99	0.9	0.98	0.92	0.97
82	0.87	0.99	0.91	1	1.07	1.05	1.01	1
83	1	0.99	0.99	1	0.9	1.02	0.99	1.01
84	0.88	0.89	0.8	0.84	1.12	0.87	1.04	0.91
85	1	1	1	1	0.99	1	0.97	1.01
86	1	0.98	0.99	1	0.98	0.86	1.06	1.01
87	1	1	1	1	1.02	0.9	1.02	1.01
88	0.95	0.99	0.88	1	0.94	0.98	0.96	1

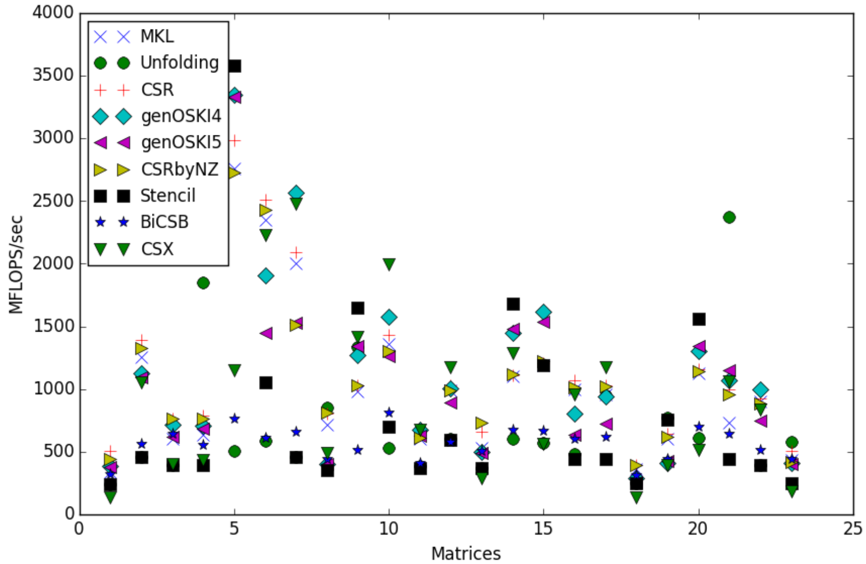

Figure A.1: MFLOPs/sec for loome2 for all methods.

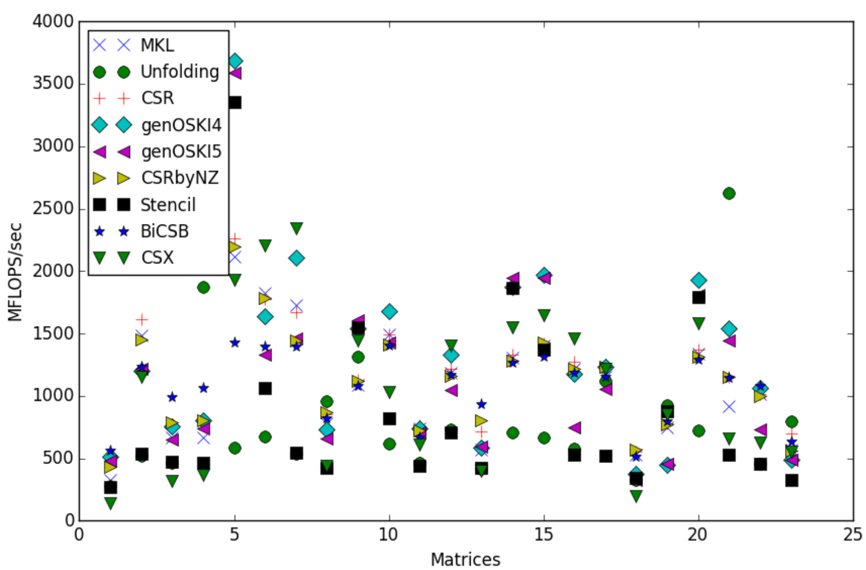

Figure A.2: MFLOPs/sec for loome3 for all methods.

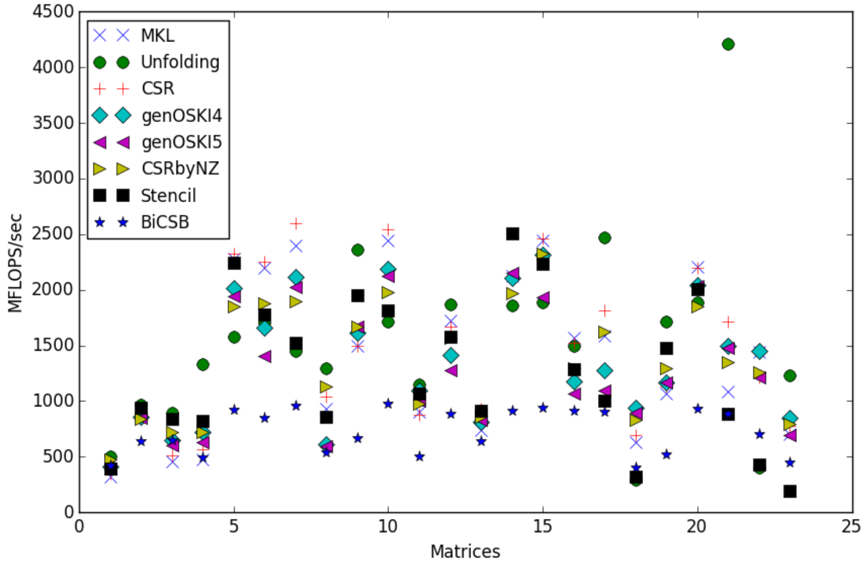

Figure A.3: MFLOPs/sec for i2pc3 for all methods.

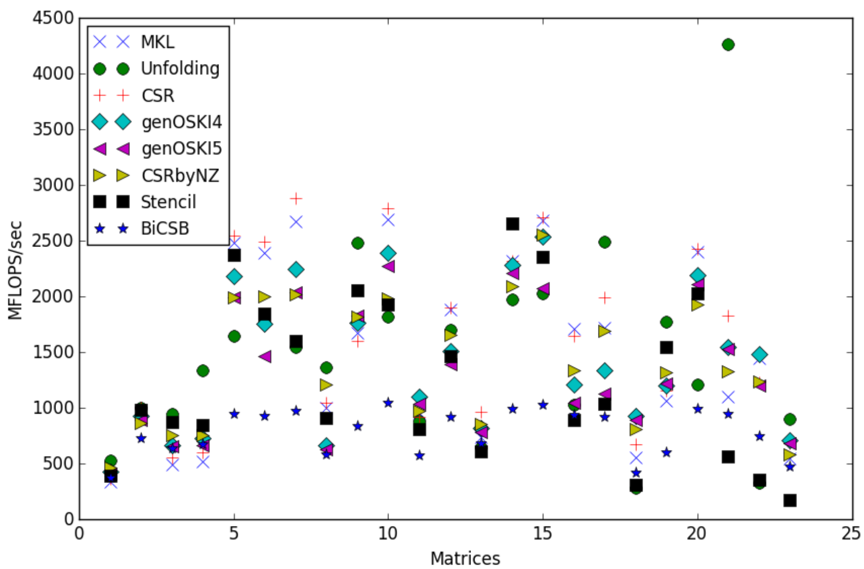

Figure A.4: MFLOPs/sec for i2pc5 for all methods.

References

[1] W. Taha and M. Nielsen, "Environment classifiers," in *POPL '03*, pp. 26–37, 2003.

[2] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha, "Mint: Java multi-stage programming using weak separability," in *PLDI '10*, pp. 400–411, 2010.

[3] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta, "Static analysis of multi-staged programs via unstaging translation," in *POPL '11*, pp. 81–92, 2011.

[4] R. Davies and F. Pfenning, "A modal analysis of staged computation," in *POPL '96*, pp. 258–270, 1996.

[5] 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, pp. 147–154, 2013.

[6] J. Mellor-Crummey and J. Garvin, "Optimizing sparse matrix vector multiply using unroll-and-jam," *International Journal of High Performance Computing Applications*, vol. 18, no. 2, 2004.

[7] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient smvm kernels," in *Proc. of ICS*, pp. 100–109, 2009.

[8] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, 2004.

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

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

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

[12] "Data for SpGEMM." http://lukeo.cs.illinois.edu/spgemmdata/.

[13] L. N. O. Steven Dalton, Nathan Bell, "Optimizing sparse matrix-matrix multiplication for the gpu," tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign, 2013.

[14] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proc. IPDPS*, 2011.

[15] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: An extended compression format for spmv on shared memory systems," in *Proc. of PPoPP*, pp. 247–256, 2011.

[16] "CSB library." http://gauss.cs.ucsb.edu/ aydin/csb/html/index.html.

[17] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. of SPAA*, pp. 233–244, 2009.

[18] "CSX library." https://github.com/cslab-ntua/csx.

[19] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self Adapting Linear Algebra Algorithms and Software," *Proc. of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005.

[20] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply," in *Supercomputing '02*, p. 26, 2002.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. of Supercomputing*, pp. 38:1–38:12, 2007.

[22] N. Bell, S. Dalton, and L. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[23] A. Logg, K.-A. Mardal, and G. N. Wells, "Automated solution of differential equations by the finite element method (chapter 6)." https://bitbucket.org/fenics-project/fenics-book/downloads.

[24] R. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimizations of Sofware and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[25] M. Frigo, "A Fast Fourier Transform Compiler," in *PLDI '99*, pp. 169–180, 1999.

[26] M. Püschel and et al., "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[27] X. Li, M. J. Garzarán, and D. Padua, "Optimizing Sorting with Genetic Algorithms ," in *CGO '05*, pp. 99–110, 2005.

[28] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *ICCS'05*, pp. 99–106, 2005.

[29] A. Jain, "poski: An extensible autotuning framework to perform optimized spmvs on multicore architectures, ms thesis," 2008.

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

[31] "V8 JavaScript Engine." http://code.google.com/p/v8/.

[32] J. Carette and O. Kiselyov, "Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code," *Science of Computer Programming*, vol. 76, no. 5, pp. 349 – 375, 2011.

[33] B. Aktemur, J. Jones, S. Kamin, and L. Clausen, "Optimizing marshalling by run-time program generation," in *GPCE '05*, pp. 221–236, 2005.

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

[35] A. Cohen, S. Donadio, M. J. Garzarán, C. Herrmann, O. Kiselyov, and D. Padua, "In search of a program generator to implement generic transformations for high-performance computing," *Science of Computer Programming*, vol. 62, no. 1, pp. 25–46, 2006.

[36] S. Kamin, L. Clausen, and A. Jarvis, "Jumbo: Run-time Code Generation for Java and Its Applications," in *CGO '03*, pp. 48–56, 2003.

[37] F. Smith, D. Grossman, G. Morrisett, L. Hornof, and T. Jim, "Compiling for template-based run-time code generation," *J. of Functional Programming*, vol. 13, no. 3, pp. 677–708, 2003.