Universiteit Leiden
Opleiding Informatica

Vectorized Sparse Matrix Kernels

using Hybrid Data Layouts

Name: Jos Zandvliet

Date: 15/12/2016

1st supervisor: Kristian Rietveld
2nd supervisor: H. A. G. Wijshoff

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The operation of multiplying a matrix with a vector is important in many fields. A method of increasing the performance of this operation is discussed in this thesis. A hybrid data layout was developed, which divides a matrix into smaller parts based on the sparsity patterns of the data. This way, the components can be multiplied more efficiently than when they are stored in a generic format. A framework program was written to test this approach on several matrices, which were converted to the hybrid format by hand. Furthermore, the matrix kernels each had both a SISD and a vectorized SIMD variant. Experiments show that while individual kernels are on par with those of an existing matrix-vector multiplication library, the hybrid format does not yield the desired performance gain. The use of SIMD instructions in the kernels improves performance, but not as much as a simple loop unroll does.

Contents

Chapter 1

Introduction

Sparse matrices often occur in linear algebra. A system of linear equations or a linear transformation can be expressed as a matrix; in many cases this matrix will be sparse. In particular, the operation of multiplying a (square) sparse matrix with a vector is important in iterative solving methods[1, §4.3.2].

Because of its importance, there is a desire to speed up this operation, as evidenced by the large amount of research on this topic. This thesis examines an approach to this speedup. The general idea is to use a hybrid data layout; this layout divides the original matrix into smaller components, where each component uses a data format that is most suited to store it.

A framework was written to load sparse matrices and perform vector multiplications on them. This framework supports several existing sparse matrix formats. Furthermore, it supports matrices in the experimental hybrid data layout. The time necessary to complete the multiplication was measured and compared with existing methods.

The first chapter will give a brief overview of sparse matrices and sparse matrix storage formats. The second chapter details the implementation of the hybrid data layout and framework program. The third and fourth chapters document the comparison experiments that were performed. The final chapter is a conclusion and discusses possible future work.

Chapter 2

Sparse matrices and storage formats

A sparse matrix is a matrix in which the majority of the entries are zero. By making use of this property, these matrices can be stored and operated upon more efficiently than if they were handled like a dense matrix.

A matrix-vector multiplication, for instance, can be greatly improved by only considering nonzero values. Since zero-values do not contribute to the result of the operation, they do not need to be stored or multiplied.

2.1 Matrix-vector multiplication

A fundamental operation on a matrix is the matrix-vector multiplication[3, §1.4]. For an $m \times n$ matrix A and n-length vector $\mathbf{b}$, if $A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix}$ and $\mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$, then the matrix-vector product $A\mathbf{b}$ is defined as:

$$A\mathbf{b} = \begin{pmatrix} a_{1,1} \cdot b_1 & \cdots & a_{1,n} \cdot b_n \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot b_1 & \cdots & a_{m,n} \cdot b_n \end{pmatrix}$$

This thesis will only consider the case in which A is a square matrix (matrices where $m = n$).

2.2 Existing sparse matrix formats

There are several formats to store sparse matrices. These formats can be divided into those that can store arbitrary matrices (such as the COO format) and those that can only store a specific subset of sparse matrices (such as the CDS format).

Each format requires its own sparse matrix/vector multiplication routine, also called a 'kernel'. The formats listed here are those that have a kernel implementation in the framework. Except for the Block-Compressed Diagonal format these are the most commonly used sparse matrix formats, with kernel implementations in nearly all major linear algebra libraries.

To illustrate the formats, a matrix of $n \times n$ with m nonzeros will be shown as it would be stored in each format.

2.2.1 Coordinate

The simplest format is the coordinate (COO) format[4, §3.4], in which the row, column, and value of each nonzero entry is stored in a separate array, each of length m.

Consider the following example matrix A:

$$A = \begin{pmatrix} 8 & 0 & 0 & 6 & 7 & 0 \\ 5 & 3 & 0 & 0 & 0 & 9 \\ 6 & 5 & 5 & 0 & 3 & 6 \\ 0 & 2 & -1 & 0 & 4 & 0 \\ 0 & 7 & 0 & -4 & 8 & 3 \\ -6 & 0 & 0 & 0 & 4 & 7 \end{pmatrix}$$

This matrix would be stored as the arrays:

`row`	1	1	1	2	2	⋯	5	5	6	6	6
`column`	1	4	5	1	2	⋯	5	6	1	5	6
`value`	8	6	7	5	3	⋯	8	3	-6	4	7

2.2.2 Compressed Sparse Row

In the coordinate format, rows with more than a single value will create long runs of the same row number in the `row` array. The compressed sparse row (`CSR`) format[2, §1][4, §3.4] compresses these runs, which reduces the space needed for the `row` array from m entries to $n + 1$ entries. It achieves this compression by storing the index at which each row starts rather than storing each row number explicitly. The other two arrays are stored as in the coordinate format.

The `row`-value of a row `i` is set to the index of the first nonzero entry in that row. If there are no nonzero entries, it it set to equal the value of `row[i+1]`. This allows for the calculation of the amount of entries in a row `i` by taking `row[i+1] - row[i]` when using zero-based indexing.

A variant of CSR that stores column indexes rather than row indexes, called Compressed Sparse Column[2, §2], exists but was not implemented as it has no major advantages over CSR.

The example matrix A would be stored as the arrays:

row_start	0	3	6	···	21						
column	1	4	5	1	2	···	5	6	1	5	6
value	8	6	7	5	3	···	8	3	-6	4	7

2.2.3 Compressed Diagonal Storage

The compressed diagonal storage format[2, §4] is specifically tailored to the storage of matrices that consist only of values along the diagonal. It stores, as a dense matrix, the values of the diagonal and those within a certain (matrix-specific) left and right bandwidth.

Consider the following example matrix B:

$$B = \begin{pmatrix} 8 & 6 & 0 & 0 & 0 & 0 \\ 7 & 5 & 3 & 0 & 0 & 0 \\ 0 & 0 & 9 & 6 & 0 & 0 \\ 0 & 0 & 5 & 5 & 3 & 0 \\ 0 & 0 & 0 & 6 & 4 & -2 \\ 0 & 0 & 0 & 0 & 9 & 4 \end{pmatrix}$$

Using a left bandwidth of 1 and a right bandwidth of 1, B would be stored as

value	0	8	6	7	5	3	···	6	4	-2	9	4	0

Note that the first and last value, whose locations are outside the boundaries of the matrix and thus do not correspond to a matrix element, are stored as 0.

2.2.4 Block-Compressed Diagonal

This format, called Block-Compressed Diagonal (BCD), was developed after noticing that some of the sparse matrices used for the experiments contained one or more diagonals made up of dense blocks (see Figure 2.1).

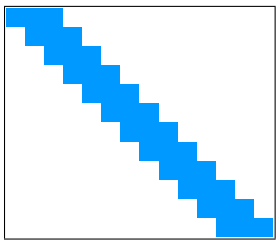

Figure 2.1: An example of a (partial) block diagonal from `benelechi1`. The blocks are 18×6 with a stride of 6.

BCD resembles the compressed diagonal storage; all blocks are stored as a dense matrix. Unlike CDS, BCD does not support diagonals that fall partially outside the boundaries of the matrix.

There are existing formats that make use of dense blocks, such as Block-Compressed Row Storage[2, §3]; this format allows for arbitrary placement of blocks of equal size, which requires additional arrays to store the location of each block.

Consider the following example matrix C:

$$C = \begin{pmatrix} 8 & 6 & 0 & 0 & 0 & 0 \\ 7 & 5 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 9 & 6 & 0 & 0 & 0 \\ 0 & 0 & 5 & 5 & 0 & 0 \\ 0 & 0 & 3 & 6 & 0 & 0 \end{pmatrix}$$

The dense blocks of C are 2×2 with a stride of 1. Using these parameters, C would be stored as

`value`	8	6	7	5	3	0	$\cdots$	5	5	3	6

Note that a block-compressed diagonal of height 1 and stride 1 is essentially equivalent to the compressed diagonal storage (excepting values outside the matrix as mentioned earlier).

Chapter 3

Quilt matrices & Implementation of multiplication kernels

The idea of the hybrid storage format is to divide a matrix into multiple components, and using the most suited data format to store and multiply each component. A framework was implemented in C99 to perform matrix-vector multiplication on matrices in arbitrary storage formats.

In most cases, the (preferred) matrix storage format is derived automatically from the component signature (further explained in Section 3.3), but this signature can be overridden through a command-line option.

Matrices are loaded in from files in the Matrix Market exchange format[5]. These files are ASCII-encoded text files consisting of header containing information about the matrix in question, including but not limited to whether or not it is a symmetric, in which case only the lower triangular portion is specified; or what kind of data it consists of (e.g. real or complex numbers). This header line is followed by optional comments, each preceded by the percent sign. Finally, a line specifies the width, height, and the amount of nonzeros in the matrix, followed by the actual data points.

3.1 Vector processing

As described in Section 2.1, a matrix/vector multiplication is essentially a large number of scalar multiplications, one for each matrix element. Because these multiplications are independent of each other, they can be performed in parallel.

Vector processing is computing using instructions that work on multiple values at a time; these are also known as single-instruction multiple-data or SIMD instructions. Values are loaded into a vector register. A vector register can hold multiple values of a given type, but the amount

of values it can hold depends on the bit length of the values' data type and the bit length of the register. For example, a 256-bit register can contain four double-precision floating point numbers, which are 64-bit each.

Intel's Advanced Vector Extensions (AVX) is one of the instruction sets that provide SIMD instructions. AVX was later expanded to AVX2. Notable of AVX2 is that it provides a gather instruction, but lacks a scatter instruction, which has to be performed serially. A gather operation retrieves values from an array at given indexes, a scatter operation writes values to an array at given indexes. Since this is the reverse of gather, and since these operations will most often appear together, it is unusual that only one of the two is offered by the AVX instruction set. To use AVX instructions in a program in a higher-level language, such as C, Intel offers 'intrinsic instructions'[9], which are function wrappers that correspond with AVX assembly instructions.

As an example, an excerpt is given of the implementations of a program that sums the entries in an array of doubles. The SISD variant simply takes each entry and adds it to the total. The SIMD variant, written using AVX intrinsics, sums every 4-length subvector, and sums the values of the resulting vector. Because the array might not be a multiple of 4, remaining values need to be added using SISD instructions. The SIMD variant requires at most $\lfloor \frac{\texttt{ARRAY\_LENGTH}}{4} \rfloor + 3$ additions, while the SISD variant always requires `ARRAY_LENGTH` additions.

SISD sum

```
double total = 0;
int idx = 0, n = ARRAY_LENGTH;
for(; n > 0; n--, idx++)
  total += array[idx];
```

SIMD sum

```
double total = 0;
// a __m256d is a vector of 4 doubles
__m256d sum, tmp;

int idx = 0, n = ARRAY_LENGTH;
for(; n >= 4; n -= 4, idx += 4){
  // _load_pd loads four consecutive doubles from memory into a _mm256d
  tmp = _mm256_load_pd(&array[idx]);
  // _add_pd adds __m256d's element-wise
  sum = _mm256_add_pd(sum, tmp);
}

double *z = (double *)&sum;
total = z[0] + z[1] + z[2] + z[3];

for(; n > 0; n--, idx++)
  total += array[idx];
```

3.1.1 Gather operation

In vector processing, the gather operation retrieves a vector of values from an array based on an array index given for each value. The gather is used in the implementation of the COO and CSR kernels to collect the values from the input vector based on column indexes.

For an input vector `in`, an vector of column indexes `column`, the gather operation is performed as follows:

```
double out[4];
out[0] = in[column[0]];
out[1] = in[column[1]];
out[2] = in[column[2]];
out[3] = in[column[3]];
```

In AVX2, there is a dedicated instruction to gather 4 doubles from memory, `_mm256_i32gather_pd`. Both forms of gather were considered when performing the experiments.

3.2 Hybrid representation format: 'quilt'

The hybrid layout is named a 'quilt matrix', so named because of the similarity to a quilted blanket. Like the blanket, a quilt matrix is composed of several possibly overlapping 'patches'; in this case, component matrices. An example of a quilt matrix can be seen in 3.1; the top image shows the original matrix, and the bottom image shows how this matrix could be divided into smaller components.

A quilt matrix consists of its dimensions and a list of all of its components. Each of these components consists of matrix data and a coordinate. This coordinate represents the displacement of the top-left corner of the component from the $(1, 1)$ position in the entire quilt matrix. Within the matrix data the component's size, amount of nonzeros, whether it is symmetric or not, the data format of the entries and a pointer to that data is stored.

Quilt matrix components are each stored in a separate Matrix Market file. The coordinates in each component have to be translated to compensate for the displacement in the quilt matrix. This adjustment is performed during the division into components. The different components are tied together using an index file. This file contains

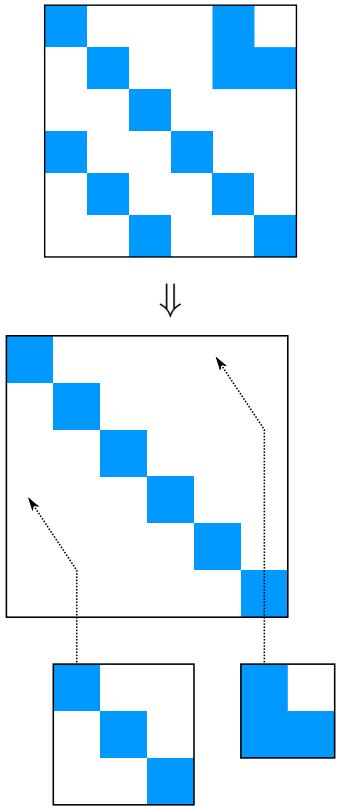

Figure 3.1: A single 6×6 matrix (above) is split up into components (below). The blue squares each represent a nonzero. The original placement of the components is kept track of.

the information required to locate and place all components and thus compose the entire quilt matrix. It begins by specifying the size of the quilt matrix and the amount of components it has, followed by a list of the components as a file name and displacement coordinate pair.

There are no guards in place to make sure that any given coordinate has only one value in one component matrix. Because the different components of the quilt matrix can be possibly overlapping, care must be taken that a value that appears in one component is not repeated in another.

A matrix/vector multiplication on a quilt matrix is done by sequentially multiplying each component with the appropriate subvector. Because each component contains information of what data format its entries have been loaded in, the multiplication can be delegated to the appropriate multiplication kernel for that data format. The result of the multiplication of each component is placed into an intermediate vector as large as the result vector. This was necessary because the Intel MKL kernels clear the given result vector before multiplication, overwriting any earlier data. The resulting intermediate vectors are summed to obtain the final result.

However, when the used multiplication kernels do not clear the result vector beforehand and do not access the result vector for rows where there are no entries, the intermediate vector is not needed, and the quilt matrix can be multiplied with zero copy operations.

It is also possible to multiply multiple vectors with the same quilt matrix at the same time. When handling each component, all vectors are multiplied against it one by one. This way, when doing multiple matrix-vector multiplications, the matrix does not need to be read from memory for each multiplication but can remain in the cache, which is more efficient.

When multiplying a component, the multiplication kernels only have access to the component's corresponding subvector. Because of this, symmetry of values is only supported within components themselves; if an entire component needs to be mirrored across the 'global' diagonal, a different subvector is required but the kernel does not have access to it. To store a symmetric matrix as a quilt matrix, the mirror images have to be added manually by adding additional components.

3.3 Component signature

The framework extends Matrix Market files with a 'component signature', a line which describes the preferred storage format of the data contained in the file. The signature is placed directly following the Matrix Market header.

The signature consists of several parts:
`%$` *type* [*parameters*]

The signature begins with the sequence `%$`, which is considered by other Matrix Market applications to be a comment due to the initial percentage sign, but is detected by the framework.

The mandatory type field is a three-letter code identifying the preferred storage format. The following type codes are recognized:

`DNS`	dense matrix
`COO`	coordinate
`CSR`	compressed sparse row
`CDS`	compressed diagonal storage
`BCD`	block compressed diagonal

The CDS and BCD formats require additional parameters. CDS requires the left and right bandwidth of the diagonal and the stride (the amount of columns moved right after each row). BCD requires the width and height of each block and the stride (the amount of columns moved right after each block).

Chapter 4

Experimental Setup

To compare the performance of the quilt matrix concept compared to existing data formats, the framework program loads matrices in the Matrix Market format, and performs a multiplication on them with the increasing vector $v = \begin{pmatrix} 1 \\ 2 \\ \vdots \\ n \end{pmatrix}$, of which the elapsed time is measured. The time needed to load matrices in memory is excluded from measurement.

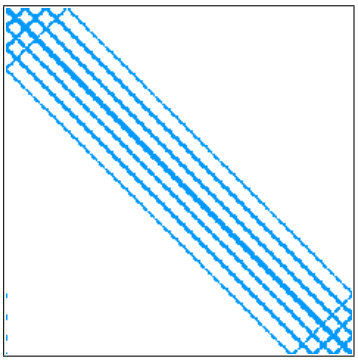

Figure 4.1: `twotone` displayed at a low resolution.

A number of matrices were retrieved from the University of Florida's Sparse Matrix Collection[6], all of them with a size greater than $100,000 \times 100,000$. The matrices that were used were the following:

- `BenElechi/BenElechi1`
- `QLi/largebasis`
- `Fluorem/RM07R`
- `ATandT/twotone`

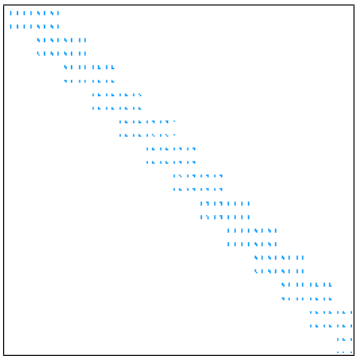

Figure 4.2: An excerpt of `twotone` at a higher resolution.

All matrices were sorted by coordinate beforehand to facilitate loading them into CSR format directly.

These matrices were then divided into several components by hand. Because this conversion was done manually, the focus was mainly on large and easy to extract structures,

such as straight diagonals. This resulted in quilt matrices of a handful of components, often leaving more chaotic structures in a 'rest' component.

The choice of which structures to extract and the preferred format to store them in was primarily based on the arrangement of the nonzeros of the structure.

As an example, the matrix `twotone` appears at first glance to be a simple collection of 7 (block) diagonals (Figure 4.1). However, zooming in on these diagonals reveals that the blocks they are made of are themselves very sparse, consisting of many tiny 16-nonzero diagonals (Figure 4.2). This sparseness makes the diagonals as a whole unsuited for storage as BCD. Each individual diagonal might be well suited for storage as CDS, however the sheer amount of these small diagonals makes it infeasible to separate them by hand.

To evaluate the performance of the quilt matrix concept, the program was run on both the original version and its quilted counterpart. All experiments were performed on one of the nodes of the Distributed ASCI Supercomputer 5 (DAS-5). These nodes contain 2.40GHz Intel® Xeon® E5-2630 v3 CPUs, which implement the AVX2 instruction set. The compiler used was GCC 5.2.0 at the O2 optimization level. The L3 cache size on each node is 20480 KiB.

The multiplication is done using each of the three modes: SISD, SIMD (vectorized), or using the sparse BLAS routines found in the Intel Math Kernel Library[7, §2]. The Intel MKL library can be set up to perform multiplications either sequentially or using multithreading. The sequential version of the library was used for a fairer comparison with the self-written routines, which are also sequential.

The time it takes to perform the multiplication is calculated using the one of the timers offered by Linux, `CLOCK_PROCESS_CPUTIME_ID`, which is a measurement of the absolute CPU time elapsed in the current process. The elapsed time is calculated from the difference between a measurement of this timer before and after the multiplication.

Furthermore, the matrix storage format of the matrix or its components was varied. For the original matrices, only the formats COO and CSR were measured. For the quilted matrices, either each component was stored as COO, or each component was stored as CSR, or each component was stored based on its embedded signature (See Section 3.3).

While the multiplication is a deterministic process, the timing of two runs may differ caused by factors including but not limited to task scheduling, invocation of CPU sleep state, or CPU frequency decreases to avoid overheating [8].

To diminish the influence of this noise, each experiment was run 16 times, in which each matrix was multiplied with a vector 1000 times. The highest and lowest 3 results were then discarded, and the remaining 10 results were averaged. Furthermore, the tool `taskset` was used to make sure the framework program was not migrated across CPUs.

Chapter 5

Results

This section will present a description, the results, and a discussion of the different experiments that were run. The first experiment will compare the SISD, unrolled SISD, and SIMD variants of the multiplication kernels. Second, two different forms of the gather operation are compared. Finally, the overhead and effectiveness of the quilt format itself are examined.

All experiments were run 16 times with 1000 repetitions. In all tables below, the average of the median 10 runtimes in seconds is shown.

5.1 Comparison of multiplication kernels

As has been described in Chapter 4, two variants of each kernel have been written: a SISD variant and a SIMD variant. Furthermore, an additional SISD variant of the kernel was unrolled 4 times. Since the SIMD variant works with vectors of 4 values, the unrolled SISD variant and the SIMD variant perform the same amount of work in a single loop iteration.

In the following table, the runtime of the different kernels on the `rm07r` matrix is shown.

	SISD	SISD unrolled	SIMD	Intel
COO	102.450 s	103.626 s	101.699 s	122.406 s
CSR	102.571 s	44.712 s	44.996 s	39.151 s

As can be seen, unrolling the SISD loop gives a large performance gain in the CSR kernel, but not in the COO kernel. The reason for this is that the unrolled CSR kernel requires only one read and one writeback to the result vector, since all four entries are in the same row. The COO kernel, however, requires a read and a writeback for each of the four entries.

Similarly, the SIMD variant of the COO kernel is only a small improvement over the SISD variants. For the CSR kernel, SIMD does improve on the SISD variant, but not as much the unrolled SISD variant.

Any variant of the COO kernels is faster than the Intel's COO kernel. The runtime of both the unrolled SISD variant and the SIMD variant CSR kernel are close to that of Intel's CSR kernel. The performance of our kernels can thus be considered to be on par with existing kernels.

5.2 AVX2 versus manual gather

To compare the two different gathers, as described in Section 3.1.1, the runtime of the COO and CSR kernels was measured on `rm07r`, in both single matrix as quilt format, using both methods of gather.

		AVX2 gather	manual gather
single matrix	COO	111.290 s	101.740 s
	CSR	45.117 s	67.840 s
quilt matrix	COO	105.480 s	90.678 s
	CSR	55.431 s	65.617 s

Using AVX2 gather gives a significant performance gain on the CSR kernel. For the COO kernel, no significant difference is seen on the single matrix; on the quilt matrix, the manual gather is faster. This is likely related to the memory access pattern of COO (accesses to distinct rows as opposed to only one row), as was seen in the previous section.

5.3 Overhead of the quilt format

A matrix in the quilt format has an overhead incurred by the need to place component results in an intermediate vector (see Section 3.2). This intermediate vector needs to be zeroed before and summed to the total result vector after the matrix-vector multiplication on each component.

To characterize the baseline of this overhead, the runtime of a single matrix is compared to quilt matrix with a single component (the same matrix). The size of the intermediate vector iv is given; it is equal to the height of the matrix. The results for the unrolled SISD variant are shown.

		single matrix	single component quilt matrix (zero-copy)	increase
`largebasis`	COO	11.416 s	12.258 s	0.842 s
$\lvert iv\rvert = 440020$	CSR	7.612 s	8.457 s	0.845 s
`rm07r`	COO	103.646 s	104.268 s	0.622 s
$\lvert iv\rvert = 381689$	CSR	44.811 s	45.517 s	0.706 s
`benelechi1`	COO	19.170 s	19.668 s	0.498 s
$\lvert iv\rvert = 246874$	CSR	17.562 s	18.000 s	0.438 s
`twotone`	COO	2.534 s	2.760 s	0.226 s
$\lvert iv\rvert = 120750$	CSR	1.555 s	1.703 s	0.148 s

This overhead stays fairly consistent. Since the overhead is independent of the kernel(s) used for multiplication, the overhead stays approximately the same between the COO and CSR kernel. The larger matrices require a larger intermediate vector and thus spend more time clearing it. The time spent copying depends on the size of the components. Note that a quilt with multiple components will have a larger overhead since the intermediate vector needs to be zeroed before and summed after *every* component multiplication.

When quilt multiplication is done with the zero-copy method described in Section 3.2, the overhead is neglible. The speed decreases are most likely caused due to the noise in the measurements, as discussed in Section 4.

		single matrix	single component quilt matrix (zero-copy)	increase
`largebasis`	COO	11.383 s	11.377 s	-0.006 s
	CSR	7.600 s	7.613 s	0.013 s
`rm07r`	COO	103.856 s	103.649 s	-0.207 s
	CSR	44.906 s	44.780 s	-0.126 s
`benelechi1`	COO	19.170 s	19.170 s	0.000 s
	CSR	17.585 s	17.547 s	-0.038 s
`twotone`	COO	2.616 s	2.565 s	-0.051 s
	CSR	1.565 s	1.560 s	-0.005 s

5.4 Effectiveness of the quilt format

The intention of the quilt format is to store each component in a format suited for the nonzero structure of that component. For each matrix, regular matrix multiplication was compared to quilt matrix format with each component in either COO or CSR format, and quilt format with each component in its preferred 'signature' format. Furthermore, a comparison with the Intel MKL library is given.

The runtimes given are of the best kernel and method (SISD or SIMD) for each format.

`largebasis`: 440020 $\times$ 440020, 5560100 nonzero, 12 components.

Single matrix	7.606 s	CSR SISD
Quilt matrix	12.452 s	CSR SISD
Quilt matrix (zero-copy)	8.320 s	CSR SISD
Quilt with signatures	12.906 s	sig SIMD
Quilt with signatures (zero-copy)	8.997 s	sig SIMD
Intel	6.661 s	CSR Intel

Even though the runtime increase incurred by the (zero-copy) quilt format is small, the choice of signatures does not succeed in improving on the single matrix approach; in fact, using signa-

tures is detrimental to the runtime, as the signature quilts are slower than their all-CSR counterparts.

`rm07r`: 381689 × 381689, 37464962 nonzero, 18 components.

Single matrix	44.887 s	CSR SISD
Quilt matrix	57.723 s	CSR SISD
Quilt matrix (zero-copy)	49.666 s	CSR SISD
Quilt with signatures	82.656 s	sig SISD
Quilt with signatures (zero-copy)	73.987 s	sig SISD
Intel	39.258 s	CSR Intel

Comparing the two zero-copy quilts, the signature quilt is considerably worse. Furthermore, as with `largebasis`, the signature quilts are worse than their all-CSR counterparts. This means that, on average, the selected formats for each component were not as efficient as the CSR format for that component.

`benelechi1`: 245874 × 246874, 669185 nonzero (12535837 nonzero with explicit symmetry), 5 components.

Single matrix	17.516 s	CSR SISD
Quilt matrix	20.492 s	CSR SISD
Quilt matrix (zero-copy)	18.181 s	CSR SISD
Quilt with signatures	21.120 s	sig SISD
Quilt with signatures (zero-copy)	18.836 s	sig SISD
Intel	13.618 s	CSR Intel

`benelechi1` is a symmetric matrix. Because the quilt format does not support external symmetry, explicit flipped components have to be added. This causes the amount of nonzeros to almost double. Even though it has to multiply almost twice the amount of nonzeros, the zero-copy quilt matrix performs only slightly worse than the single matrix. The choice of signatures for this matrix only degrades this result.

`twotone`: 120750 × 120750, 1224224 nonzero, 9 components.

Single matrix	1.558 s	CSR SISD
Quilt matrix	3.033 s	CSR SISD
Quilt matrix (zero-copy)	2.216 s	CSR SISD
Quilt with signatures	3.021 s	sig SISD
Quilt with signatures (zero-copy)	2.222 s	sig SISD
Intel	1.146 s	CSR Intel

`twotone` was originally split into block diagonals, but since the blocks are very sparse, this was not a feasable format (the block diagonal format stores each block as dense). The split into components remained, but all were set to CSR. The split incurs an overhead due to the

quilt format. The signature quilt format does not improve the result since the sparsity of the components means no use could be made of internal patterns.

When suitable kernels are used, the zero-copy method of multiplying a quilt matrix has a runtime comparable to that of the single matrix since it has little overhead.

The ability to improve performance using signature quilts depends on the choice of the signatures for each component. In these example matrices, the choice of signatures was done by hand based on the sparsity patterns of the components. With these specific signature choices, there is no improvement in the performance compared to a quilt with components in a generic format such as CSR.

5.5 Multiple vectors and component size

As mentioned in Section 3.2, a possible way to improve performance on multiplication with multiple vectors is to perform the multiplications for the different vectors consecutively for the same component. The idea behind this is that after performing the multiplication for this component for the first vector, all data for the component will reside in cache, benefitting performance of the multiplication for the subsequent vectors.

To test this, one of the components of `rm07r` was taken separately. The component itself is 334094×334094, with 4789410 nonzeros. When stored in the COO format, this component is $\approx$ 74834 KiB, too large to fit into the cache of the target machine (20480 KiB). This component was then divided into 8 equal parts (each approx. 9354 KiB).

This matrix, in its different divisions, was then multiplied with 4 vectors simultaneously (as described in Section 3.2) with 10,000 repetitions.

	absolute time	avg. time per vector
single matrix	425.875 s	106.469 s
Quilt of single component	461.014 s	115.253 s
Quilt of single component (zero-copy)	426.664 s	106.666 s
Quilt of 8 components (zero-copy)	413.349 s	103.337 s

When each component fits into the cache, the multiplication is faster, compared to both the quilt format and the single matrix format. In the case of the 8-component quilt matrix, all components fit in the cache, however only a slight performance improvement is obtained.

Using `perf` to measure the amount of cache misses, the 'approximate event count' metric shows the amount of last level cache accesses for loads more than halves, indicating that better use is made of higher level caches.

	1 component	8 component
LLC-loads	2841M	1091M
LLC-load-misses	726M	252M
LLC-stores	1.7M	1.4M
LLC-store-misses	0.1M	0.05M

The miss rate shows a small decrease for both loads and stores, going from 26% to 23% for loads and from 17% to 4% for stores. Since all components fit in the cache, it would be expected that the miss rate would go down greatly, but this does not occur.

Chapter 6

Conclusions and Discussion

In this thesis a framework was described for matrix-vector multiplication on matrices composed of multiple components. The framework is able to use a variety of kernels; kernels for several sparse matrix formats were implemented.

Of each kernel, a SISD and a SIMD variant was developed. For the COO kernel, using SIMD improves the runtime, but not significantly. The CSR kernel does improve when SIMD instructions are used, but unrolling the SISD kernel yields a slightly larger improvement than that.

Test matrices were devided by hand into multiple components to make better use of internal patterns of the nonzero values. However, it appears that the choice of formats for the components was poor, as using these 'signature' formats only degraded performance. The use of a generic format for all components gives a better result, but it was still less efficient than the single-matrix approach in most cases.

One way to improve the performance of the kernels is to make better use of the cache. Currently, all kernels work on one vector at a time, so that when multiplying a matrix (component) with a vector, the entirety of the multiplication needs to be completed before the next vector can be multiplied.

To aid this cache usage, it would be possible to adapt the multiplication kernels to accept multiple vectors and multiply them truly simultaneously, rather than having to wait for one vector's multiplication to finish. This would also improve cache behavior, since matrices larger than the cache size would not need to be separately loaded in for each vector; the matrix data would only be replaced when it has already been multiplied with all of the vectors.

Matrix components are currently multiplied sequentially. Since component multiplications are independent of each other, another possible improvement would be to perform these in parallel using multithreading.

The matrices used in this thesis were converted to the quilt format by hand. Many components

could be further split into even smaller components to make better use of the nonzero layout. To split up a matrix to such a degree would result in some cases in thousands, if not millions, of separate components; this would be infeasible to do manually. Automated detection and extraction of components could possibly be performed by existing computer vision and pattern recognition algorithms.

Bibliography

[1] Barrett, R.; Berry M.; Chan T. F. et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, Second Edition* (1994).

[2] Dongarra, J. *Sparse Matrix Storage Formats*, in *Templates for the Solution of Algebraic Eigenvalue Problems: A Practival Guide* (2000).

[3] Lay, D. *Linear Algebra and Its Applications, Third Edition* (2006).

[4] Saad, Y. *Iterative Methods for Sparse Linear Systems, Second Edition* (2003).

[5] Boisvert, R. F.; Pozo, R.; Remington, K. A. *The Matrix Market Exchange Formats: Initial Design*. NISTIR 5935 (1996).

[6] Davis, T. A.; Hu, Y. *The University of Florida Sparse Matrix Collection*. ACM Transactions on Mathematical Software 38, no 1 (2011): 1–25. Retrieved from `http://www.cise.ufl.edu/research/sparse/matrices`

[7] Intel Corporation. *Intel® Math Kernel Library Developer Reference* (2015). Retrieved from `https://software.intel.com/en-us/intel-mkl-support/documentation`

[8] Intel Corporation. *Optimizing Performance with Intel® Advanced Vector Extensions* (2014). Retrieved from `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf`

[9] Intel Corporation. *Intel® Intrinsics Guide*. Retrieved from `https://software.intel.com/sites/landingpage/IntrinsicsGuide`