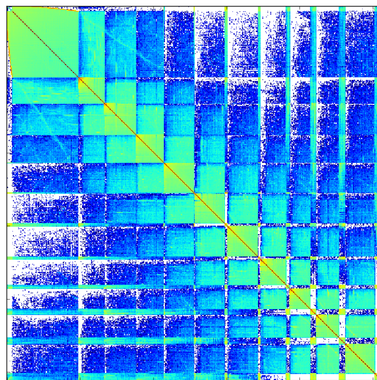
Universiteit Utrecht

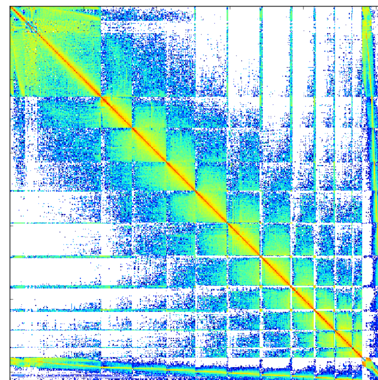
Bachelor Thesis in Mathematics

Cache optimization for sparse matrix-vector multiplication

Author:
P. J. Mulder

Supervisor:
R. H. Bisseling





Department of Mathematics
Utrecht University

June 25, 2015

Abstract

In this thesis we introduce a cost measure to compare the cache-friendliness of different permutations of the rows and columns of a given matrix. And we implement a simple algorithm that tries to reorder the rows and columns of a given matrix in hopes of increasing the cache-friendliness.

1 Introduction

With the advent of the computer, the complexity of mathematical models has grown beyond what is humanly possible. One of the tools which is often used in these mathematical models are matrices. So, unsurprisingly, since the last century the size of these matrices has grown exponentially, into the millions of rows and columns.

The size of these matrices comes with all sorts of problems. For instance, we simply cannot store these matrices by naively storing every element. Doing so for a matrix containing 32-bit integers with five million rows and columns would require $5 \cdot 10^6 \times 5 \cdot 10^6 \times 32$ bits or approximately 100 TB of memory. Fortunately, these matrices commonly have relatively few non-zero elements (i.e. they are *sparse*), which enables us to use a more efficient storage scheme such as *compressed row storage* (CRS). We explain this data structure in section 3, which is based on the works [1] and [2].

However, this leads to another problem. Suppose you are handed an enormous sparse matrix with millions of rows and columns. You are asked to multiply this matrix, by hand, with some *dense* (i.e. almost all entries non-zero) vector. Of course you can skip the zero entries of the matrix, and, luckily for you, the matrix is sparse so there are a lot of entries to skip. But few non-zeros often means that they are scattered throughout the matrix. This results in you having to constantly jump back and forth through the input and output vector as you perform the matrix-vector multiplication. Ideally, we would want all the non-zeros of the matrix corresponding to the same part of the vectors to be close together. That way we minimize the jumping back and forth and we can keep the relevant part of the input vector in our short-term memory.

For computers this works the same way, except the short-term memory of a computer is called its *cache*. For a more detailed explanation of caches see section 2, which is based on a paper by Intel [3].

Rereading the title of this thesis should give you an idea of what we are trying to achieve here: we want to transform a given sparse matrix in such a way that non-zero entries of the matrix are clustered together. To this end we present a cost measure for the cache-friendliness of a matrix and an algorithm to optimize the cache utilization of a matrix. We introduce this measure in section 4.

There already exist some algorithms for cache optimization of sparse matrices, of which the one introduced by Yzelman and Bisseling in [2] is an example. Although this algorithm produces good results, it is computationally expensive. Therefore the algorithm is only useful if the resulting matrix is reused very

often. We aim to create an algorithm which produces less cache-friendly results but does so in a timely manner. This algorithm is presented in section 5.

2 Cache

Currently processing speeds of computers are not limited by the CPU but by the latency of accessing main memory [4]. Simply increasing the speed of the main memory is not an option because fast memory is hugely expensive. Instead, computers have small chunks of fast, but expensive, memory sitting in front of the main memory. These chunks are called *caches*. Cache is used to store recently accessed parts of the main memory. If the CPU needs some data it first checks if the data is already in the cache, before loading it from main memory. If the requested data is in the cache it is called a *cache hit*, otherwise we call it a *cache miss*. If a cache miss occurs the CPU needs to read from main memory, when doing so the data read gets stored in the cache.

These caches form a hierarchy, with the slowest but biggest cache at the top — i.e. closest to the main memory — and the fastest but smallest cache at the bottom — i.e. closest to the CPU. Consequently, we also refer to these individual caches as *cache levels*. The bottom cache is referred to as the Level1-cache (or L1 for short) and subsequent levels are called L2, L3, etc. Our benchmark CPU — an Intel Core i5 3570k — has three levels of cache. Every cache reads data from the cache one level up, hence the data in L1 is a subset of the data in L2, and so on. The CPU tries the cache levels in order until it finds the data it needs.

A cache is divided into l cache lines, which are the units in which the cache loads and evicts data. These cache lines are often bigger than the data requested, so a request also loads some adjacent data. This leads to contiguous memory access barely incurring cache misses, and thus being extremely fast. In this paper we try to approximate contiguous memory access while performing sparse matrix-vector multiplication.

Because the size of the cache is several orders of magnitude smaller than the size of the main memory we need a replacement policy to decide which cache line to evict when the cache is full.

One of the simplest approaches is the *direct-mapped cache* which simply assigns every entry in main memory to the cache line with index $i \mod l$ where i is the index of the block in main memory. Then when loading a new block from memory it is clear which cache line to replace, because every entry in memory only maps to one cache line. But suppose we alternatingly need data from two blocks in memory which map to the same cache line. Then this approach is extremely slow, because the two blocks would constantly evict each other from the cache.

A better option is the *k-way associative cache* which partitions the cache lines into sets of size k, where k is a power of two. This type of cache assigns to every block in memory the set with index the least significant bits of the memory address (i.e. the index of the block in memory). That way addresses which

are close in memory get mapped to different sets, again speeding up contiguous memory access. Then, when loading a new block from memory, we replace the least recently used (LRU) cache line in the set assigned to that block.

Note that the 1-way associative cache is just the direct-mapped cache. It is not hard to see that when k gets bigger we have less chance of inducing unnecessary cache misses — because we allow more choice in which entry to evict. So why not set $k = l$? Suppose $k = l$ then the CPU does not know upfront which of the k cache lines contains the requested block, so it needs to scan all k entries. For big k this is too much overhead, and reducing k actually increases performance. Hence k is a parameter which needs to be balanced between reducing cache misses and preventing scanning.

Our test CPU consists of four (one cache per core) 8-way associative L1 caches with 64 sets, four 8-way associative L2 caches with 512 sets and one 12-way associative L3 cache with 8192 sets. The cache line size is 64 bytes, resulting in 4×32 KB L1, 4×256 KB L2 and 6 MB L3 cache.

3 Compressed row storage

As mentioned in the introduction, it is often inefficient to store every element of a sparse matrix. Much better storage formats are available, of which compressed row storage (CRS) is an example. CRS works by maintaining three separate arrays: nzs, which contains the values of non-zero elements, col_ind, which holds the column indices for these non-zero elements and row_start which consists of indices of the previous two arrays corresponding to row boundaries.

Given an $m \times n$ matrix A, with $nz(A)$ the number of non-zeros in A, we need only $2nz(A)+m$ entries in memory. Using our previous example of a matrix with five million rows and columns and assuming that only 0.001% of the elements are non-zero (i.e. $nz(A) = 250 \cdot 10^6$) we need only $\left(2 \times 250 \cdot 10^6 + 5 \cdot 10^6\right) \times 32$ bits of memory, which amounts to about 2 GB. A huge improvement over the previous 100 TB.

Although CRS is somewhat more complex than naive storage, the algorithm for matrix-vector multiplication stays relatively simple, see Algorithm 1.

Algorithm 1 Matrix-vector multiplication for CRS matrices.

Input:
nzs, col_ind and row_start for a sparse matrix A;
the dimensions m and n of A;
and a dense input vector x of length n.

Output:
a dense output vector y of length m with $y = Ax$.

1: Allocate y of size m and set $y = \mathbf{0}$.
2: **for** $i = 0\,;\, i < m\,;\, i \mathrel{+}= 1$ **do**
3:     **for** $k = row_start[i]\,;\, k < row_start[i+1];\, k \mathrel{+}= 1$ **do**
4:         $j := col_ind[k]$
5:         $y[i] \mathrel{+}= nzs[k] * x[j]$
6: **return** y

4 A measure for cache-friendliness

In this section we introduce a cost measure for the cache-friendliness of a sparse matrix with regards to matrix-vector multiplication. To this end we first model a single level of cache. Next, we analyse this model to obtain a measure. Finally, we extend this measure to multiple levels of cache, and to the cache-oblivious case.

4.1 Single cache

As mentioned before we can iterate over contiguous structures without unnecessary cache misses. Assume sparse matrices are stored in a contiguous way — which is the case when using CRS. Then sparse matrix-vector multiplication can always be done in such a way that all unavoidable cache misses occur while accessing the input and output vectors. Therefore the storage format of the matrix completely determines the access pattern of the input and output vectors, and, subsequently, the cache performance of the matrix-vector multiplication.

If we define a graph G with the vertices given by the non-zeros of a matrix, and we require G to be complete, the access pattern corresponds to a *Hamiltonian path* in this graph — i.e. a path that visits every node exactly once. The cache performance is thus a property of this path. We will try to approximate this property by analyzing a simple cache model.

Notice that when a matrix A is stored in CRS format the rows are accessed in order and thus do not incur unnecessary cache misses. Therefore, we are only interested in the column access pattern of these matrices. For simplicity we assume all our matrices use the CRS storage format. However, the algorithm and formulas in this section can easily be adapted to the general case by adding the cost of A and A^T.

Let us assume a direct-mapped cache with l cache lines of size s. Let the sequence $(a_i)_{0 \le i < nz(A)}$ be the column access pattern of a given matrix A. Care-

fully reading the description of the direct-mapped cache in section 2 results in Algorithm 2 for the number of cache misses.

Algorithm 2 Direct-mapped cache model

Input:

a sequence $(a_i)_{0\leq i<k}$ of memory addresses;
l the number of cache lines and their size s.

Output:

the number of cache misses c.

1: $c := 0$
2: $f_0(j) := -\infty$ for all $0 \leq j < l$ ▷ $f_i(j)$ is the memory address of the contents of cache line j at step i, the value -1 corresponds to an empty cache line.
3: **for** $i = 0\,;\, i < k\,;\, i \mathrel{+}= 1$ **do**
4: $\quad mli := \lfloor a_i/s \rfloor$ ▷ memory line index
5: $\quad cli := mli \bmod l$ ▷ cache line index
6: $\quad$ **if** $f_i(cli) \neq mli$ **then** ▷ cache miss
7: $\quad\quad c \mathrel{+}= 1$
8: $\quad f_{i+1}(j) := \begin{cases} mli & \text{if } j = cli, \\ f_i(j) & \text{otherwise.} \end{cases}$ ▷ update cache
9: **return** c

This algorithm can be compressed into the expression

$$c(l,s) = \sum_{i=0}^{k-1} \begin{cases} 1 & \text{if } \operatorname{prev}(i,l,s) = -\infty \text{ or } a_i \neq a_{\operatorname{prev}(i,l,s)} \\ 0 & \text{otherwise,} \end{cases}$$

where

$$\operatorname{prev}(i,l,s) = \max\left\{ j' < i \,:\, \lfloor a_i/s \rfloor \equiv \lfloor a_{j'}/s \rfloor \pmod{l} \right\},$$

and the maximum of the empty set is defined as $-\infty$.

4.2 Multiple caches

Suppose we have N caches with cache line size s and suppose the I-th cache has l_I cache lines. To measure the cache-friendliness of the access pattern with respect to the whole memory hierarchy we simply sum all the cache misses multiplied by a cache-specific penalty p_I:

$$c_{\text{specific}} = \sum_{I=0}^{N-1} p_I \cdot c(l_I, s). \tag{1}$$

Recall that the data in lower caches is fully contained in higher caches. Therefore, we only need to find the last level of cache where a cache miss occurs; every

cache below this level has a cache miss and every cache above it has a cache hit. This way (1) simplifies to:

$$c_{\text{specific}} = \sum_{i=0}^{k-1} \sum_{I=1}^{L(i)} p_I, \tag{2}$$

where $L(i) = \max \left\{ I \,\middle|\, \text{prev}(i, l_I, s) = -\infty \text{ or } a_i \neq a_{\text{prev}(i, l_I, s)} \right\}$.

For our test CPU we have $N = 3$, with $s = 64$ bytes, $l_1 = 8 \cdot 64$, $l_2 = 8 \cdot 512$, $l_3 = 12 \cdot 8192$ and $p_1 = 6$, $p_2 = 30$ and $p_3 = 380$ cycles.

4.3 Cache-oblivious

But suppose we do not know the layout of the memory hierarchy, or we need to optimize for a heterogeneous environment. We still want to be able to measure cache efficiency. Not knowing the parameters of our caches we need to take both small and large caches into account. One idea is to introduce several virtual caches growing exponentially. Note that the penalty for missing a cache increases with its position in the memory hierarchy, and its position increases with its size. Hence, we set the penalty for the virtual caches equal to their size.

So suppose A has K columns we introduce $N = \lfloor \log_2(K) \rfloor$ virtual caches with the parameters $p_I = l_I = 2^I$. We assume a constant cache line size s (e.g $s = 64$ bytes — the most common cache line size in use today).

Substituting these parameters into (2) results in:

$$c_{\text{oblivious}} = \sum_{i=0}^{k-1} \sum_{I=1}^{L(i)} 2^I = \sum_{i=0}^{k-1} 2^{L(i)+1} - 2$$

5 Increasing cache-friendliness

In this section we introduce an algorithm for improving the cache-friendliness of a matrix. Several of these algorithms already exist, such as the algorithms given by Yzelman and Bisseling in [2] and [5]. But these algorithms are either computationally expensive (and thus result in net profit only after a large number of multiplications), or have less pronounced gains in cache efficiency (which achieves net profit sooner, but saves less time in the long run). With our algorithm we we aim for the middle ground; we try to obtain a net profit sooner than the computationally expensive algorithms but still produce a better permutation than the faster algorithm.

As noted in section 4 the access pattern, and subsequently the cache performance, of the matrix is determined by its storage format. Because we cannot change the storage format the only way to improve cache performance is by changing the matrix itself. One of the least destructive changes we can do is permuting the rows and columns of the matrix. The output of our algorithm is therefore a permutation of rows P_r and a permutation of columns P_c such

that the input matrix A under these permutations (i.e. $P_r A P_c$) becomes cache-optimized.

Because the cache-optimized matrix is structurally different from the original matrix we cannot simply substitute this matrix for the original matrix. Luckily matrix-vector multiplication stays relatively simple (we only need to permute the input and output vectors):

$$Av = {P_r}^{-1}(P_r A P_c)({P_c}^{-1} v).$$

The algorithm we present here actually consists of a more general framework:

Given a matrix A, interpret the rows of A as the nodes of a complete graph $G(V, E)$. The weight of the edges between these nodes will be given by a certain function, $V \times V \to \mathbb{R}_{\geq 0}$, which measures the similarity between rows. Next compute an access pattern by approximating a maximum Hamiltonian path (i.e. a Hamiltonian path of maximum total weight). This access pattern then induces a permutation of rows P_r. The permutation of the columns P_c is obtained exactly the same way but with A replaced with its transpose A^T. The matrix $P_r A P_c$ is now in a cache-optimized form. See Algorithm 3 for the pseudocode.

Algorithm 3 CacheOptimizeMatrix

Input:
a sparse matrix A, stored in CRS format.

Output:
P_r, P_c permutations of rows, columns of A, s.t. $P_r A P_c$ is cache-optimized

1: G_r := RowSimilarityGraph(A)
2: *row_path* := GreedyMaxHamiltonianPath(G_r)
3: P_r := PathToPermutation(row_path)

4: G_c := RowSimilarityGraph(A^T)
5: *col_path* := GreedyMaxHamiltonianPath(G_c)
6: P_c := PathToPermutation(col_path)

7: **return** P_r, P_c

5.1 Similarity of vectors

For our algorithm we need a way to measure how similar two vectors are with regards to the cache. Furthermore, this similarity measure should be invariant under permutations such that its value does not change when we switch from permuting rows to permuting columns. Because our goal is a fast algorithm we choose the inner product of the corresponding *sparsity pattern vectors* — i.e. exactly the same vector but with the non-zeros replaced by 1 (since the cache does not care about the specific value of an element). Note that this is similar to the cosine similarity and the Jaccard index, except it does not take into account the magnitude of the vectors. Doing so would be better, but for the sake of simplicity and performance we decided not to do so. Using this measure we

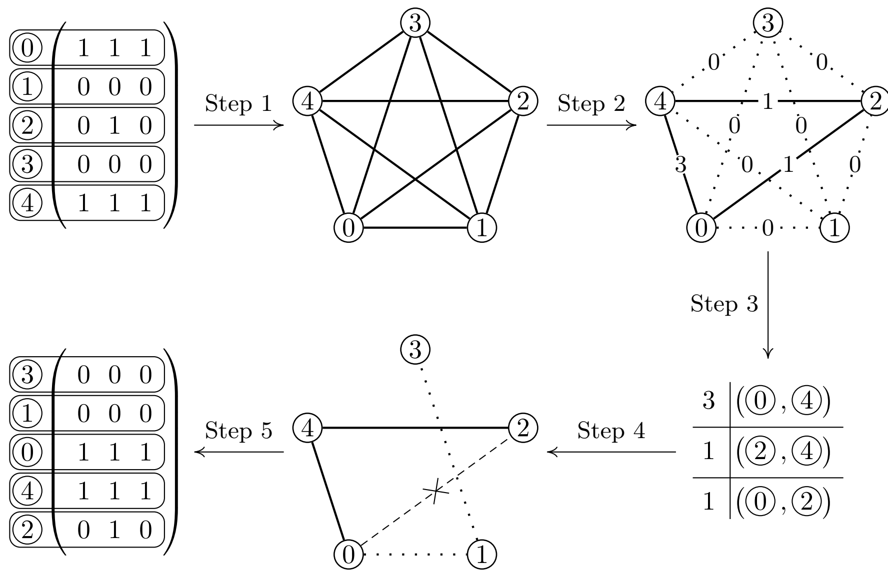


Figure 1: A schematic representation of Algorithm 3 using a simple example matrix. Step 1: interpret the rows of the matrix as vertices of a complete graph. Step 2: use the inner products of the rows as edge weights, do not store zero weight edges – indicated by the dotted lines. Step 3: sort the stored edges by weight. Step 4: construct a Hamiltonian path by adding the edges in descending order while ensuring you do not add an edge which induces a cycle – see edge $(0, 2)$ – or a vertex with degree > 2. Then join all components and add all missing vertices – indicated by the dotted lines. Step 5: compute the permutation induced by the Hamiltonian path. The result is a cache-optimized permutation of the rows of the original matrix.

build the complete graph with the vertices given by the rows of the matrix and the edge weights by the inner product. Note that the inner product can be zero. If this is the case we do not store the edge, we know by its absence that it has zero weight. See Algorithm 4 for the algorithm.

Algorithm 4 RowSimilarityGraph

Input:
a sparse $m \times n$-matrix A in CRS format: *col_ind* and *row_start*.

Output:
a complete graph $G(V, E)$ where V consists of all row indices and the edge weights are given by the similarity of the endpoints. Edges of weight zero are not stored.

```
1: inner_products := SparseMatrix()
2: m0 := 0                                              ▷ row index 0
3: for i0 = 0 ; i0 < nz(A) ; i0 += 1 do
4:     if i0 = row_start[m0 + 1] then                   ▷ end of row:
5:         m0 += 1                                      ▷ so increase row index

6:     m1 := row_start[m0 + 1]                          ▷ row index 1
7:     for i1 = m1 ; i1 < nz(A) ; i1 += 1 do

8:         if i1 = row_start[m1 + 1] then               ▷ end of row:
9:             m1 += 1                                  ▷ so increase row index

10:        if col_ind[i0] = col_ind[i1] then            ▷ both rows have a non-zero:
11:            if (m0, m1) not in inner_products then
12:                inner_products[m0, m1] := 0
13:            inner_products[m0, m1] += 1 ▷ so add one to the inner product

14: V := {0, . . . , m − 1}
15: E := {edge (m0, m1) with weight w : (m0, m1, w) ∈ inner_products}
16: return G(V, E)
```

5.2 Greedy maximum Hamiltonian path

We also need a way to approximate a maximum Hamiltonian path. Note that the existence of a maximum Hamiltonian path is guaranteed by the fact that the graph is complete. There are some existing algorithms for approximating the maximum Hamiltonian path problem (MaxHPP). These are based on approximating a maximum Hamiltonian cycle — also called the maximum travelling salesman problem (MaxTSP). We can reduce MaxHPP to MaxTSP by constructing a new graph by adding a node u to the original graph and adding zero weight edges from u to all other vertices. A maximum Hamiltonian cycle in this new graph corresponds to a maximum Hamiltonian path in the original graph [6]. Two example algorithms are a deterministic algorithm by Serdyukov [7] and a randomized algorithm by Hassin and Rubinstein [8]. These algorithms

compute a maximum Hamiltonian path with, respectively, total weight at least 3/4 of the maximum, and a expected total weight of at least 25/33 times the maximum.

However, both these algorithms run in $O(|V|^3)$ time [9], which conflicts with our goal of constructing a fast algorithm. Instead, we use a greedy algorithm to approximate the maximum Hamiltonian path. This algorithm can be implemented in $O(|E|\log(|E|)) = O(|V|^2\log(|V|))$ time — depending on which sorting algorithm is used.

The algorithm is relatively simple. First sort all the **stored** — i.e. non-zero weight — edges in the graph. Next, construct a Hamiltonian path by continually adding the edge with maximum weight which does not induce a cycle or a node with degree bigger than 2. We continue this process until there are no more edges left. Note that zero weight edges are never added to the path, therefore the edges necessary to connect two components might be missing. Furthermore, suppose there is a non-zero which is the sole non-zero in its row and column, then the inner product of this row with any other row is zero. Subsequently, the corresponding vertex has only zero weight edges in the graph, and hence can not be part of a component.

To fix both issues simply merge all components in arbitrary order and append all vertices which have not yet been added to a component to the path. The resulting path is a Hamiltonian path with the total weight an approximation of the maximum. See Algorithm 5 for a pseudocode version of this algorithm and Figure 1 for a schematic overview of the complete algorithm.

Algorithm 5 GreedyMaxHamiltonianPath

Input:
a complete undirected graph $G(V, E)$, in which if an edge is not stored it is assumed it weight is zero.

Output:
a Hamiltonian path $\{e_i\}_{0 \leq i < |V|-1}$ which approximates the maximum Hamiltonian path.

▷ a component is a maximal set of connected edges within our partial Hamiltonian path

1: $C :=$ Map() ▷ components indexed by their leaf – or outer – nodes
2: $closed_nodes := \{\}$ ▷ nodes of degree 2, or inner nodes
3: **for** $e \leftarrow$ Sorted(E) **do** ▷ i.e. the **stored** edges
4: $(v_0, v_1) := e$ ▷ v_0 and v_1 are the nodes of edge e

5: **if** v_0 or v_1 in $closed_nodes$ **then** ▷ either not a leaf node:
6: **continue** ▷ would introduce a node of degree > 2

7: **if** both v_0, v_1 not in C **then** ▷ both not part of an existing component:
8: $c := \{e\}$ ▷ so add a new component
9: point $C[v_0]$ and $C[v_1]$ to c

10: **else if** v_0 in C, but v_1 not in C **then** ▷ one is part of a component:
11: $C[v_1] \leftarrow C[v_0]$ ▷ so extend that component
12: delete v_0 from C
13: add e to component $C[v_1]$
14: add v_0 to $closed_nodes$

15: **else if** v_1 in C, but v_0 not in C **then** ▷ same as previous but swapped
16: $C[v_0] \leftarrow C[v_1]$
17: delete v_1 from C
18: add e to component $C[v_0]$
19: add v_1 to $closed_nodes$

20: **else if** $C[v_0] = C[v_1]$ **then** ▷ both are in the same component
21: **continue** ▷ would introduce a cycle

22: **else** ▷ they are in different components:
23: join components $C[v_0]$ and $C[v_1]$ into c by using e ▷ so join them
24: delete v_0 and v_1 from C
25: $(v_2, v_3) =$ LeafNodes(c)
26: point $C[v_2]$ and $C[v_3]$ to c
27: add v_0 and v_1 to $closed_nodes$
28: add all leaf nodes to $closed_nodes$
29: join all components in C arbitrarily and call the result HP
30: append all nodes in $V/closed_nodes$ to HP ▷ zero weight edges are not stored
31: **return** HP

6 Results

We tested our algorithm on seventeen sparse matrices from The University of Florida Sparse Matrix Collection [10], which are listed in Table 1. See Figure 2 for a sparsity plot of some of these matrices and their cache-optimized reordering.

Our test results are based on multiplying the matrices with the 1-vector. During this test we measured the elapsed CPU-time and simulated the cache misses — using *cachegrind* [11]. We also computed the cache-specific and cache-oblivious cost measure from section 4. The ratio of the results between the original and cache-optimized matrix are presented in Table 2.

Looking at these results we notice four things:

- our algorithm almost always improves the L1 miss ratio;
- the L3 miss ratio stays roughly the same, although there are some outliers in both directions;
- the L3 miss ratio has the most impact on the time ratio;
- the cache-specific cost ratio seems to be most correlated with the L1 miss ratio;
- the cache-oblivious cost ratio gives a good indication of the time ratio.

The cache-oblivious cost seems to give a better indication of the actual running time than the cache-specific cost. This is rather surprising, but it might be a result of the fact that we used a direct-mapped cache model to derive the cost function which differs from the actual cache type used in our test CPU. The cache-oblivious cost simulates more levels of cache, this probably hides these shortcomings in our model.

The cache-specific cost shows spikes for some matrices. These matrices all resemble diagonal matrices. So they are already close to optimal. Our algorithm has trouble handling these close-to-optimal matrices, which results in a slightly wider diagonal. This wider diagonal has a big impact on the smaller caches, but less so on the bigger caches. This might explain the spike, because the cache-specific cost is more sensitive to smaller caches.

The decrease in L1 cache misses indicate that our approach is promising; at least locally the structure of the matrices is far more cache-friendly. However, L3 cache misses are by far the most expensive, therefore the L3 cache has the most influence on wall clock performance. Because of the bigger cache size the L3 cache is more sensitive to global structure than to local structure. The fact that the L3 behaviour stays roughly the same can be explained by the greedy nature of our algorithm, which leads to local structure. So, even though our algorithm increases cache-friendliness, it does not increase wall clock performance as much as we would like.

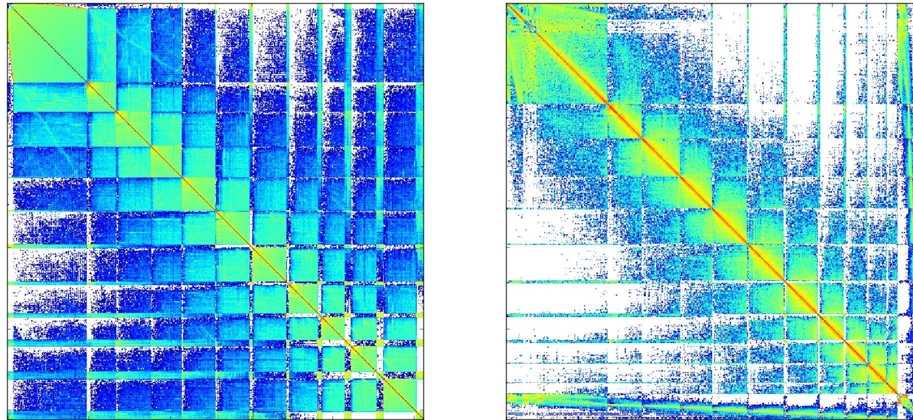
In order to decrease the number of L3 cache misses a natural way forward would be to add a global element to our approach. Recall from section 5 that we did

Name	Rows	Columns	Non zeros
fidap037	3 565	3 565	67 591
memplus	17 758	17 758	99 147
lhr34	35 152	35 152	746 972
msc23052	23 052	23 052	1 142 686
language	399 130	399 130	1 216 334
lp_nug30	52 260	379 350	1 567 800
dimacs10_144	144 649	144 649	2 148 786
18_tbdlinux	112 757	20 167	2 157 675
s3dkt3m2	90 449	90 449	3 686 223
bmw7st_1	141 347	141 347	7 318 399
G3_circuit	1 585 478	1 585 478	7 660 826
af_shell9	504 855	504 855	17 588 845
asia_osm	11 950 757	11 950 757	25 423 206
cage14	1 505 785	1 505 785	27 130 349
GL7d20	1 437 547	1 911 130	29 893 084
hugetrace-00010	12 057 441	12 057 441	36 164 358
af_shell10	1 508 065	1 508 065	52 259 885

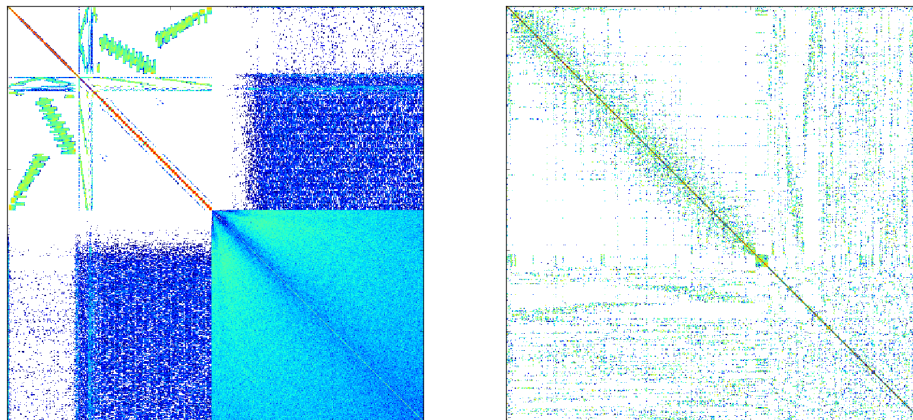
Table 1: Test matrices

not store the zero weight edges in Algorithm 4, and therefore we are left with disjoint components which need to be joined on line 29 of Algorithm 5. Our testing indicates that we are left with a modest number of such components — think thousands instead of millions. We could join these components in a particular way for a global reordering.

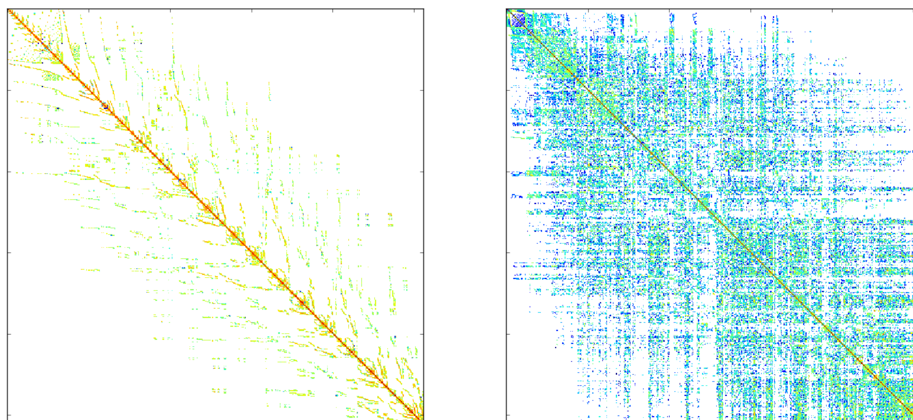
One way to do this reordering would be to choose a vector representative for each component. These representatives should give an indication of the global structure of the components. We can then use these vectors as input for another run of our algorithm, replacing the sparsity pattern vectors of rows with the representatives directly. This results in a cache-optimized ordering of the components, which likely improves the global structure. A straightforward choice for the representatives of a component would be the sum of all the sparsity pattern vectors of the rows in each component. Running an algorithm on the output of another algorithm, as we do here, is called a *multilevel* approach in the literature and has shown promising results [12].



(a) `hugetrace-00010`



(b) `dimacs10_144`



(c) `cage14`

Figure 2: Sparsity plots of several matrices. (left = original matrix, right = cache-optimized matrix)

Name	CS	CO	L1	L3	Time
fidap037	9.25	1.00	0.98	1.00	0.95
memplus	0.54	1.00	0.93	1.00	1.01
lhr34	0.66	1.00	1.06	1.00	1.00
msc23052	0.17	0.96	0.93	1.00	0.98
language	0.63	1.02	0.94	1.03	1.01
lp_nug30	0.95	0.90	0.84	0.87	1.25
dimacs10_144	0.13	0.53	0.34	1.01	0.66
18_tbdlinux	0.77	0.94	0.76	1.00	0.92
s3dkt3m2	26.28	1.00	1.00	1.00	1.07
bmw7st_1	0.52	0.99	0.98	1.00	1.03
G3_circuit	0.81	0.98	0.86	1.02	1.11
af_shell9	1.30	1.00	1.00	1.01	1.05
asia_osm	0.64	0.89	0.95	0.97	0.90
cage14	0.46	1.14	0.99	1.09	1.76
GL7d20	0.79	1.05	0.71	1.09	1.06
hugetrace-00010	0.44	0.38	0.73	0.80	0.79
af_shell10	10.32	1.01	1.00	1.01	1.08

Table 2: Ratios for the costs, cache misses and CPU time when multiplying with the 1-vector. The ratios are determined by dividing the measured values obtained from the unpermuted matrix by the values obtained from the cache-optimized matrix. (CS = Cache-specific cost, CO = Cache-oblivious cost

7 Conclusion

We introduced a cost measure for the cache-friendliness for different permutations of the rows and columns of a matrix. This measure may be used with knowledge of the cache hierarchy or without: in a cache-oblivious way. Our results indicate that the cache-oblivious measure is superior to the cache-specific measure because it hides shortcomings in the used cache model. Hence, we recommend using the cache-oblivious measure in all cases.

Next, we introduced an algorithm for increasing the cache-friendliness of a matrix. This algorithm calculates a permutation of rows and columns based on an approximation of a maximum Hamiltonian path in the complete graphs induced by the rows and columns of the matrix. The edge weights in this graph are given by a similarity measure. Though there is still plenty of room for improvement, this simple algorithm already gives very promising results.

Finally, we gave some pointers how to improve upon the algorithm. We expect that adding a multilevel level step would cause the resulting matrix to admit a global cache-optimized structure. This global structure hopefully leads to a decrease in L3 cache misses and, consequently, to improvements in CPU time.

References

[1] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, 2000.

[2] A. N. Yzelman and Rob H. Bisseling. Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.

[3] Intel. An Overview of Cache, 2007. URL `http://download.intel.com/design/intarch/papers/cache6.pdf`.

[4] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[5] A. N Yzelman and Rob H. Bisseling. A Cache-Oblivious Sparse Matrix-Vector Multiplication Scheme Based on the Hilbert Curve. In *Progress in Industrial Mathematics at ECMI 2010*, volume 17, pages 627–633, 2012.

[6] T.P. Gevezes and L.S. Pitsoulis. The shortest superstring problem. In T.M. Rassias, C.A. Floudas, and S. Butenko, editors, *Optimization in Science and Engineering*, pages 189–227. Springer New York, 2014.

[7] A.I. Serdyukov. An algorithm with an estimate for the traveling salesman problem of the maximum. *Upravlyaemye Sistemy*, 25:80–86, 1984. In Russian.

[8] R. Hassin and S. Rubinstein. Better approximations for max TSP. *Information Processing Letters*, 75(4):181–186, September 2000.

[9] A. Barvinok, E.K. Gimadi, and A.I. Serdyukov. The Maximum TSP. In G. Gutin and A.P. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, Combinatorial Optimization, chapter 12, pages 585–607. Springer US, 2007.

[10] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[11] N Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, University of Cambridge, 2004.

[12] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. (unpublished overview), December 2015. URL `http://arxiv.org/abs/1311.3144`.